UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

DEPARTAMENTO DE INGENIERÍA TELEMÁTICA

ITT: Sistemas de Telecomunicaciones

Proyecto Fin de Carrera

HTTP/2: Analysis and measurements

Author: José Fernando Calcerrada Cano

Tutor: Marcelo Bagnulo Braun

Cotutor: Anna Maria Mandalari

January 2016

“This, Jen, is the Internet.”

Maurice Moss, IT Crowd.

Abstract

HTTP/2: Analysis and measurements

by José Fernando Calcerrada Cano

The upgrade of HTTP, the protocol that powers the Internet of the people, was published as RFC on May of 2015. HTTP/2 aims to improve the users experience by solving well-know problems of HTTP/1.1 and also introducing new features. The main goal of this project is to study HTTP/2 protocol, the support in the software, its deployment and implementation on the Internet and how the network reacts to an upgrade of the existing protocol.

To shed some light on this question we build two experiments. We build a crawler to monitor the HTTP/2 adoption across Internet using the Alexa top 1 million websites as sample. We find that 22,653 servers announce support for HTTP/2, but only 10,162 websites are served over it. The support for HTTP/2 Upgrade is minimal, just 16 servers support it and only 10 of them load the content of the websites over HTTP/2 on plain TCP.

Motivated by those numbers, we investigate how the new protocol behaves with the middleboxes along the path in the network. We build a platform to evaluate it across 67 different ports for TLS connections, HTTP/2 Upgrade and over plain TCP. Considering both fixed line and mobile network, we use a crowdsourcing platform to recruit users. Middleboxes affect HTTP/2, especially on port 80 for plain TCP connections. HTTP/2 Upgrades requests are affected by proxies, failing to upgrade to the new protocol. Over TLS on port 443 on the other hand, all the connections are successful.

Keywords: HTTP/2, HTTP, HTTP Upgrade, Software support, Analysis, Adoption, Deployment, Implementation, Internet, Measurement, Middleboxes, Mobile networks, Port, Protocols, Proxies

Acknowledgements

First of all, I want to thank all my friends for their support before and during the realization of this dissertation, cheering me up, giving me the strength to finish it and for making my path much easier. My friends from Uni, my friends from my hometown, my friends from Southampton, thanks.

A special thanks to Lena Noreus, Joel Jenvey, Michał Skokowski and Lee Boyton for their time. Their feedback and corrections were really appreciated.

Thank you Karolína Hajská for your support. Your corrections, your understanding and your smile made me go forward. Without you I know it would have been much harder.

I want to thank Marcelo Barnulo for giving me this opportunity to finish my studies with a project that I have really enjoyed. I am also very grateful to Anna Maria Mandalari for all her help and support during the last months, guiding me through the process. Advice, suggestions, corrections and laughs were an exceptional and priceless help to me.

Thanks to my sister, my brothers and their partners. Even now I am a bit far way, I know they are always there for me when I need them.

And a very special thanks to my parents, for all their understanding, their endless support, their care and their love, especially in the hardest moments of my life. Without them this would never have happened. Thank you dad and mum.

Thanks to everyone.

José Fernando Calcerrada Cano

Contents

List of Figures

List of Tables

Dedicated to my parents for all their endless support…

Chapter 1

Introduction

The Internet has evolved since its creation, along with the protocols used in it. HTTP protocol is almost 16 years old and the usage of the web is not the same as it was in at the beginning.

A new version of the protocol has been published as RFC on May 2015[1]. HTTP/2 aims to improve the users' experience by solving well-known problems of HTTP/1.1. HTTP/2 in fact introduces new features like header compression, full request and response multiplexing, support for prioritization and server push.

The IETF discussed for a long time about whether HTTP/2 should be encrypted by default. On one hand, encryption provides strong privacy for end-users and encrypted streams and as proved in other protocol upgrades, it allows to traverse the network without modification with higher success. On the other hand, the use of encryption has some implications: it imposes overhead in servers as well as clients -which can be crucial in small devices-, it can degrade user performance on high latency links, if there is a requirement to inspect or cache HTTP traffic, certificates are too expensive.

Considering all of these premises, the final HTTP/2 standard defines two ways of supporting HTTP/2: HTTP/2 over TLS (H2) and HTTP/2 over clear TCP (H2C). In the case of TLS, the application protocol negotiation happens at the same time as the encryption negotiation using application-layer protocol negotiation (ALPN) extension, the string "h2" identifies the protocol. In the case of plain TCP, the negotiation uses the HTTP Upgrade mechanism. In this case, the identifier is "h2c". Support for unencrypted HTTP/2 should not be implemented on the default HTTP port, unless client and servers are well-know.

Today's Internet consists of a plethora of network entities like switches, routers, firewalls, NATs and proxies, that can alter HTTP requests and responses, especially headers, to provide their functionalities. The unpredictable and diverse behaviours of such

middleboxes can be problematic when adopting new protocols. It is unclear what the interaction between ”h2c” header Upgrade mechanism and today’s Internet ecosystem will be.

To consider HTTP/2 a successful protocol, not only the improvements must be considered, all the agents involved in the network need to support it. To answer this, we build a crawler to monitor the HTTP/2 adoption on the Internet, using the Alexa top 1 million websites as sample.

The results show significant support for HTTP/2 considering its recent introduction. 22 500 of surveyed websites announced support for it over TLS connections as of October 2015. Only 10 000 (45%) of them provide content over new protocol however.

Despite major browser vendors currently only supporting H2, several websites already support H2C. Of those 22,500 websites that announce support for H2, only 16 site have support for H2C. Motivated by these observations, in this work we set out to quantify the feasibility of H2C in today’s Internet.

Moreover, to evaluate how the new protocol behaves with the middleboxes in the network, we build a platform to evaluate it across different ports. Our methodology is as follows: we set up two servers for encrypted and unencrypted communications listening on 67 ports, supporting H2 and H2C respectively. We use all these ports in order to emulate not only regular HTTP traffic (Web, port 80/443), but also HTTP traffic used by other REST protocols like, for example, SNMP and CMIP network management (port 2301), Remote Procedure Call over HTTP (port 593), webmail HTTP service (port 8990). Next, we deploy two clients: a browser application, only for H2 communications, and an Android application using OkHttp[2]-an HTTP library with support for HTTP/2- and a custom client to test HTTP/2 Upgrade that attempt to contact our severs on those ports using both H2 and H2C. We gather multiple vantage from which to run our testing browser and Android application through Microworkers crowdsourcing platform[3] to test different networks.

Over a period of two weeks we recruit more than 650 users distributed across 38 countries; 355 users perform measurements from fixed line networks, and 322 users on mobile networks. We build a complex data-set with more than 120 000 connections.

Results show that middleboxes affect H2C deployment, especially on port 80. HTTP/2 Upgrade requests are affected by proxies, failing to Upgrade to HTTP/2. Ironically, we conclude that because of the interaction between H2C and existing middleboxes, HTTP/2 needs to be encrypted to work properly.

Chapter 2

HTTP/2

The state of the World Wide Web has changed over the last decade. Browsers have evolved from simple web page viewers to containers of complex and ambitious Web applications. The devices used to navigate the Internet have changed, with an exponential number of smartphones on the market, along with the networks used to deliver the data. This evolution and a continuous growth of the bandwidth of the networks have changed the usage of the Web: from slow file downloads to music and video streaming services, real-time video conferences with multiple users, instant messaging (on the phone), etc.

The modern Internet presents completely different challenges from those for which the protocol was originally designed. These contemporary problems can only be overcome by introducing a new protocol which is capable of providing an experience conducive to the way in which the Web is now used.

HTTP/2 is an updated version of the most widely used protocol on the Internet. It aims to solve a number of issues in HTTP/1.1. The aim is to offer a better experience in modern websites by providing an optimized transport of HTTP's semantics to an underlying connection.

In order to accomplish this, HTTP data is framed and multiplexed over the same connection in multiple concurrent streams, providing better performance, higher throughput, reduced latency, and lower resource consumption. All of this is achieved without changing any of the semantics of HTTP, but only working on the revision of the wire protocol (HTTP headers, methods, etc.).

2.1 State of the Web

HTTP is an application protocol for distributed, collaborative, hypermedia information systems, over which stands the World Wide Web. Right now it is the most used and widely adopted application protocol on the Internet. From its simple beginnings as a single line, a keyword and a document path, to fetch an hypertext document it has become the protocol not just for browsers, but for almost all Internet-connected software and hardware due to its relative simplicity and knowledge acquired about it over the last years.

The protocol became an Internet standard in 1999[4], and that is more than fifteen years ago. Now there are billions of users online, the number of Webs have increased by orders of magnitude and along with their complexity, and the HTTP specification has started to show its age and the unresolved inefficiencies that came with it.

Web sites have evolved and grown from few requests and tens of kilobytes of data transferred per page to Web applications that require, on average, about 100 requests and over 2 megabytes of data across 15 different domains, taking between 1 to 5 seconds to load. Figure 2.1 shows the evolution of the transfer size and total number of requests per page on average over the last 5 years[5]:

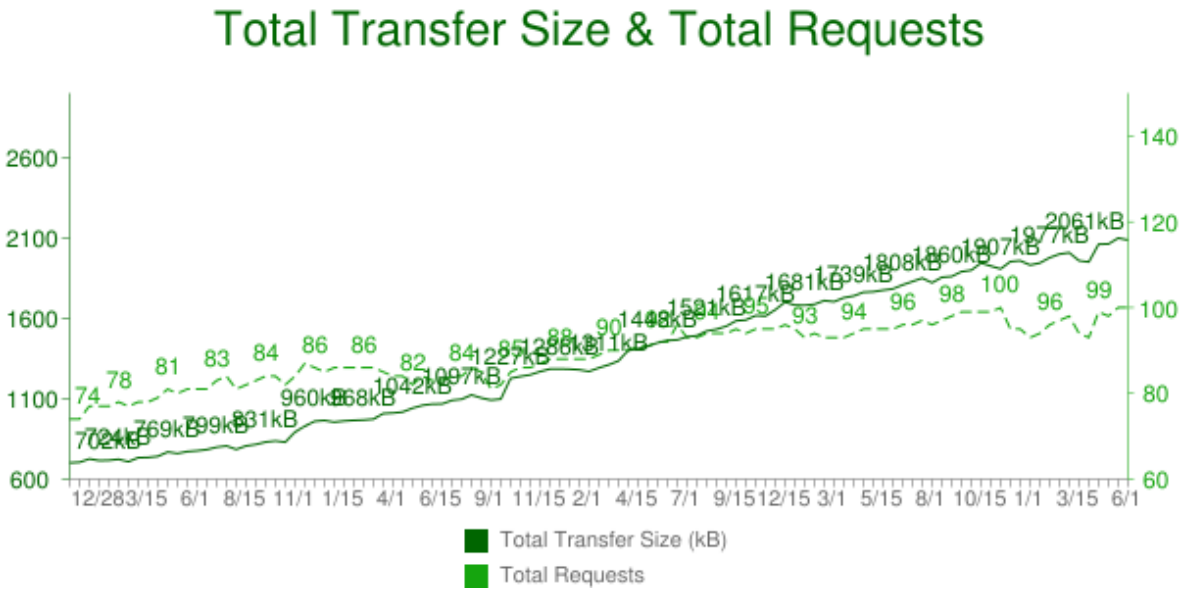

Figure 2.1: Transfer Size and Requests (HTTPArchive.org, 2010-2015)

The devices used to browse the Web have changed as well. Since the appearance of the smartphones, these have replaced phones and their number increments exponentially. Smartphones provide a new experience of the communication, a very easy way to “be connected” anywhere at any time. Figure 2.2 shows the comparison between devices

over the last 6 years collected by StatCounter[6], about 34% of the websites are served to mobile phones and 5% to tablets devices.

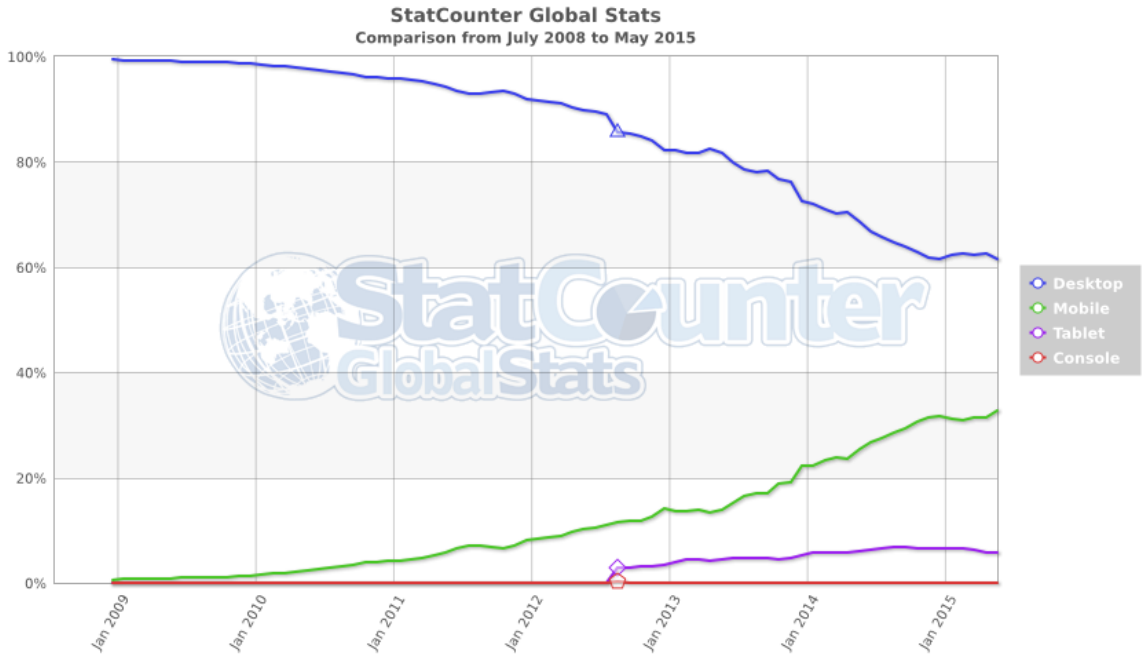

Figure 2.2: Devices usage (StatCounter, July 2015)

Smartphones introduce a different challenge for the Web and HTTP. Mobile networks have different characteristics compared to wired connections. They have a much higher and more variable latency, one of the main problems with HTTP. The bandwidth in those networks, especially the uplink, is also more constrained than in the case of wired connections. This mobility comes with another trade-off: interrupted connections. They can be caused by a number of circumstances like changes between base stations, poor coverage area, etc.

2.2 Current protocols

HTTP/1.1 protocol is not very efficient in its usage of the transport layer and it is very latency sensitive; an issue which especially affects mobile networks. HTTP does not really work as a full duplex connection, most of the time the connection is in an idle state as only one request can be placed in a connection at a time.

In order to improve the user's experience, several techniques had been used to reduce the load time of websites:

- Browsers open several connections per domain (~6 connections[7]).

- Sites can serve files from multiple domains.
- Inline data to avoid creating extra requests.
- Concatenate files to similarly reduce the number of requests.

This last two techniques also affect HTTP caching. Inline data is not cacheable and breaks resource prioritization. If a single character changes on a file then it will affect the concatenated file and thereby invalidate the cache.

These techniques have improved the overall experience for users, but they have lead to even poorer network usage, wasting lots of resources and also breaking the way HTTP is meant to work.

Pipelining is another feature of HTTP/1.1. HTTP pipelining is a technique in which multiple HTTP requests are sent on a single TCP connection without waiting for the corresponding responses. It has never been fully implemented due the difficulties related to error handling, and the fact that the head-of-line blocking issue is still there.

HTTP headers are other issue with HTTP/1.1. They are very verbose and take a considerable number of bytes to transmit, mainly because of Cookies, thus slowing down initial requests and adding a lot of overhead sending the same bytes in every request.

Another problem is that HTTP/1.1 is a text protocol, while it is easily inspected by humans it is hard for machines to parse it. Text protocols are neither efficient nor easy to implement correctly: optional white spaces, different termination tokens and other quirks make it harder to differentiate between the protocol and the payload. It is also more prone to parsing and security errors. This problem particularly affects HTTP headers, making them hard to be read by machines due their variable length.

In order to resolve all of these inherent issues, Google started to develop an experimental protocol called SPDY in 2009[8]. This protocol does not change the semantics of HTTP, it is simply a binary layer that stands between HTTP and TCP aiming to resolve the majority of the aforementioned issues. Over the years it proved to have better performance, better network usage, and demonstrated to the world that it was possible to update HTTP with a better protocol.

In fact, SPDY was starting to become the defacto standard for the web, so this lead to creation of an updated version of the protocol with SPDY at its base.

2.3 HTTP/2 overview

HTTP/2 design had few goals in mind: to improve TCP usage, be more resource friendly, reduce latency, and to keep HTTP semantics and paradigms as HTTP/1.1 is going to be around for a while.

In order to accomplish this HTTP/2 introduces a binary framing layer. Requests and responses are broken down into multiple frames and transferred between client and server in the same connection. Each tuple request-response runs in its own stream, allowing the multiplexing of several requests and responses in a single connection by interleaving frames.

These changes allow for new features like flow control and prioritization to ensure that multiplexed streams are used efficiently. Another benefit of using one single connection is that the server can push data to the client and send resources even before the client knows they will be needed.

The semantics do not change. The high-level API of the HTTP protocol remains exactly the same. Applications will continue working as before. They are not affected as clients and servers will be handling this for them. The changes are only low-level, to address the performance limitation of the protocol and add extra features.

2.4 Protocol overview

Basic unit of the protocol is the frame. HTTP messages are split and encapsulated into frames. The standard defines the following frame types:

- `DATA`: Transport requests or responses data.
- `HEADERS`: Transport header fields (they are stateful for the connection).
- `PRIORITY`: Used to sent the priority of a stream.
- `RST_STREAM`: Used to finalize streams lifetime or to cancel them.
- `SETTINGS`: Set the configuration parameters of the connection.
- `PUSH_PROMISE`: Sent by the server to push a response to the client.
- `PING`: Used to check the round-trip time and the connection state.
- `GOAWAY`: Used to notify the remote peer to do not create more streams and close the connection.
- `WINDOW_UPDATE`: Used to implement flow-control for stream and connection.
- `CONTINUATION`: Carry additional headers if they do not fit in a single `HEADERS` or `PUSH_PROMISE` frame.

There are two groups of frames: HEADERS, PUSH_PROMISE, CONTINUATION and DATA which are used mainly to send data between peers and they are associated with a stream; and PRIORITY, RST_STREAM, SETTINGS, PING, GOAWAY and WINDOW_UPDATE frames are used for flow-control, management and configuration in the connection and streams.

HTTP/2 connections are multiplexed into streams. Each stream is almost independent from other streams and usually contains a HTTP request and its response. Frames can be either associated with a specific frame or to the connection as a whole.

Once the connection has been established, either plain TCP using HTTP Upgrade mechanism or via TLS, both ends send a connection preface as a final confirmation of HTTP/2 being used. Client and server send a SETTINGS frame to indicate the configuration parameters which are not negotiated and must be acknowledged by the other end. Then the client can start sending HTTP requests to the server.

First the client sends a HEADERS frame, followed by any CONTINUATION frames if needed, with the HTTP request. As HTTP headers are stateful for the connection in HTTP/2, to avoid race-conditions, no other frames can be sent until they are sent. If there is any body in the request, DATA frames are used for this purpose. The same process will be used by the server to respond.

This happens concurrently for multiple communications on the connection. Interleaved with DATA frames, any of the control frames can appear to change the status of the connection or the streams.

If server push is enabled, the server can send a PUSH_PROMISE frame for data that will be requested by the client. This frame is basically a request sent by the server instead of the client. Once the PUSH_PROMISE frame is sent, the server can send DATA frames with the response. As PUSH_PROMISE must be cacheable, these streams can be closed by the client if the resource is not needed.

In contraposition to HTTP/1.1, HTTP/2 uses one connection for multiple requests-responses. Once the connection is not longer needed, a GOAWAY frame is sent to terminate gracefully the connection. If there is any issue during the connection and it becomes unusable, client and/or servers can close the TCP connection. They should first try to send a GOAWAY frame.

2.5 Starting HTTP/2

HTTP/2 uses the same `http` and `https` URI schemes used by HTTP/1.1. And it shares the same default ports: 80 for `http` URIs and 443 for `https` URIs.

As a result, implementations processing requests for target resource URIs are required to first discover whether the server supports HTTP/2. The means by which support for HTTP/2 is determined are different for `http` and `https` URIs.

A client that makes a request for an `http` URI without prior knowledge about support for HTTP/2 on the next hop uses the HTTP Upgrade mechanism. The client makes an HTTP/1.1 request that includes an Upgrade header field with the `h2c` token. Such request MUST include exactly one `HTTP2-Settings` header field.

The value of the `HTTP2-Settings` field is the payload of a SETTINGS frame encoded as base64url. Since the Upgrade is only intended to apply to the immediate connection, a client sending the HTTP2-Settings header field MUST also send `HTTP2-Settings` as a connection option in the Connection header field to prevent it from being forwarded.

Requests that contain a payload body MUST be sent in their entirety before the client can send HTTP/2 frames. If concurrency of an initial request with subsequent requests is important, an OPTIONS request can be used to perform the Upgrade to HTTP/2.

A server that supports HTTP/2 accepts the Upgrade with a 101 (Switching Protocols) response. After the empty line that terminates the 101 response, the server can begin sending HTTP/2 frames. These frames must include a response to the request that initiated the Upgrade. Servers must ignore an `h2` token on plain TCP.

The first HTTP/2 frame sent by the server MUST be a server connection preface consisting of a `SETTINGS` frame. Upon receiving the 101 response, the client MUST send a connection preface, which includes a `SETTINGS` frame.

A client that makes a request to an `https` URI uses TLS with the application-layer protocol negotiation (ALPN) extension. The protocol identifier in this case is `h2`. The `h2c` protocol identifier must not be sent by a client or selected by a server if TLS is used.

Once TLS negotiation is completed, both the client and the server must send a connection preface.

A client can learn that a particular server supports HTTP/2 by other means, send the connection preface (Section 3.5) and then may immediately send HTTP/2 frames to such a server; servers can identify these connections by the presence of the connection preface. This only applies to HTTP/2 connections over plain TCP.

In HTTP/2, each endpoint is required to send a connection preface as a final confirmation of the protocol in use and to establish the initial settings for the HTTP/2 connection. The client and server each send a different connection preface.

The client connection preface starts with this sequence of 24 octets:

```
PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n
```

and it is followed by a `SETTINGS` frame.

The server connection preface consists of a potentially empty `SETTINGS` frame that MUST be the first frame the server sends in the HTTP/2 connection.

The `SETTINGS` frames received from a peer as part of the connection preface must be acknowledged after sending the connection preface. To avoid unnecessary latency, clients are permitted to send additional frames to the server immediately after sending the client connection preface, without waiting to receive the server connection preface.

2.6 Streams and Multiplexing

Streams are an independent bidirectional sequence of frames exchanged between the client and server during a HTTP/2 connection. Streams have the following characteristics:

- One HTTP/2 connection can contain multiple concurrent streams, with peer interleaving frames from multiple streams.
- Streams can be established and used by both or just one endpoint.
- Streams can be closed by any of the two peers.
- The order of the frames is important. Receivers process frames in the order they are received.
- Streams are identified by a positive integer. Streams started by the client must use odd numbers, streams initiated by the server use even numbers.

Streams have an associated state, the transition between states is based on the type of frame received along with the flags set on them. The lifecycle of streams is shown in Figure 2.3.

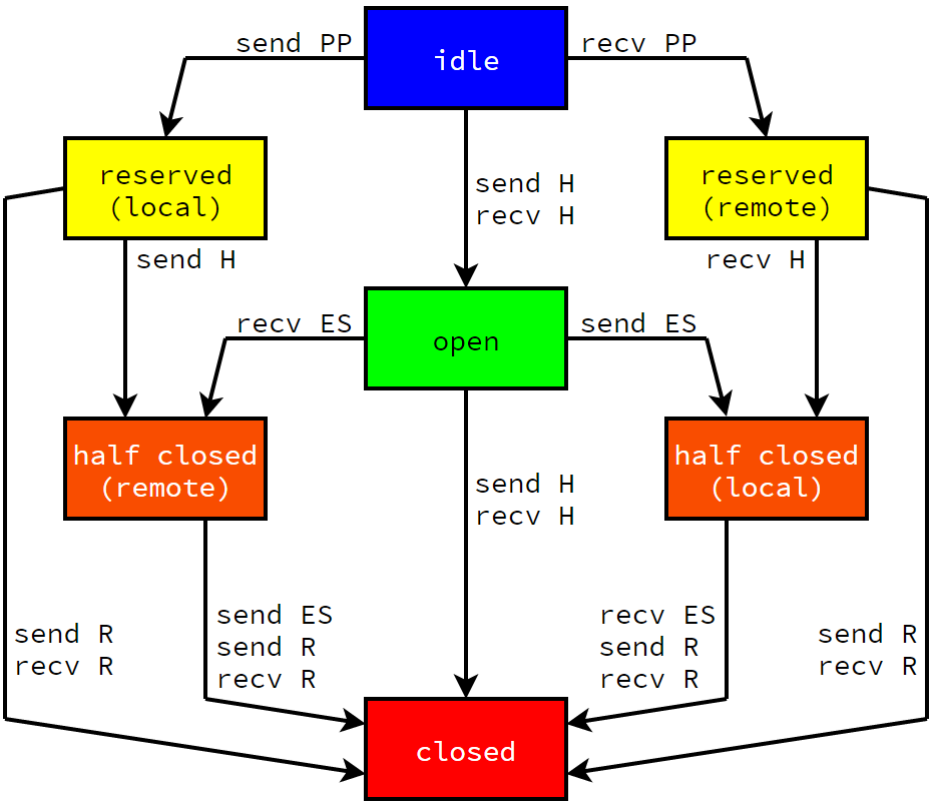

Figure 2.3: Streams lifecycle

send endpoint sends this frame

recv endpoint receives this frame

H `HEADERS` frame (with implied `CONTINUATION`s)

PP `PUSH_PROMISE` frame (with implied `CONTINUATION`s)

ES `END_STREAM` flag

R `RST_STREAM` frame

The state of the streams depends on which endpoint is considered. Due to the latency of the transmission, the state of a stream viewed from the sender can be different from the receiver until it receives the frames that changed to that state.

Streams have the following states and allowed transitions:

- **idle**: The initial state for all streams. Only `HEADERS` and `PRIORITY` frames are allowed to be sent in this state.

  The following transitions are possible from this state:

  – Sending or receiving a `HEADERS` frame changes the status to *open*.

  - Sending a PUSH_PROMISE frame on another stream reserves the stream for later use and changes the status to *reserved (local)*.
  - Receiving a PUSH_PROMISE frame on another stream reserves the stream for later use and changes the state to *reserved (remote)*.
- **reserved (local)**: A stream enters into this status after a PUSH_PROMISE frame has been sent in another stream, reserving this stream. The stream is associated with an open stream initiated by the remote peer. The allowed frames for this state are HEADERS, PRIORITY, RST_STREAM and WINDOW_UPDATE.

  The following transitions are valid:

  - Receiving a HEADERS frame change the status to *half-close (remote)*.
  - Either endpoint sends a RST_STREAM frame causes the stream to become *closed*. This releases the stream reservation.
- **reserved (remote)**: After a PUSH_PROMISE frame is received by the peer, the identified stream enters into reserved state. The frames allowed in this state are HEADERS, PRIORITY, WINDOW_UPDATE and RST_STREAM.

  In this state, the following transitions are possible:

  - Receiving a HEADERS frame changes the status to *half-closed (local)*.
  - Any endpoint can send a RST_STREAM frame to change the status of the stream to *closed*.
- **open**: Streams in this state can be used by both endpoints. Any type of frame can be sent in stream on this status.

  In this status, the following transitions can happen:

  - Sending a frame with END_STREAM flag set change the status to *half-closed (local)*.
  - Receiving a frame with END_STREAM flag set modifies the status of the stream to *half-closed (remote)*.
  - Either endpoint sent a RST_STREAM frame, the stream status changes to *closed*.
- **half-closed (local)**: peers cannot send frames in streams on this status except for WINDOW_UPDATE, PRIORITY and RST_STREAM.

  Only one transition is allowed in this status, to close status. This transition happens in two cases:

  - Sending or receiving a RST_STREAM frame.
  - Receiving a frame with END_STREAM flag set.
- **half-close (remote)**: a stream enters in this state after the sender issue a frame with END_STREAM flag set. Frames any other than WINDOW_UPDATE, PRIORITY or RST_STREAM must be treated as stream error.

Streams in this status can only change to *closed* status under the following circumstances:

  - Sending a frame with `END_STRAM` flag set.
  - Receiving or sending a `RST_STREAM` frame.

- **closed**: this is the ending state, there is no transitions allowed in this state. No more HTTP data can be sent in the stream. Only `PRIORITY` frames are allowed to be sent, any other frame must be treated as a stream error of type `STREAM_CLOSED`. `PRIORITY` frames can be sent in closed streams to prioritize streams depends on it.

2.6.1 Flow Control

HTTP/2 introduces a flow-control scheme to ensure that streams in the same connection do not interfere with each other by using all the resources and blocking the rest of the streams. Flow-control is directional and two windows are applicable for each stream: one for the stream itself and another one for the entire connection.

This scheme is based on a simple window value kept between peers on every stream that indicates the number of octets of data that the sender can issue. For each flow-controlled frame sent, the sender must decrease the value of the windows. Separated `WINDOW_UPDATE` frames are sent to update stream and connection flow-control windows.

Flow-control windows must not exceed 2^{31}-1 octets. If such case happens, a frame with an error code of `FLOW_CONTROL_ERROR` must be sent by the receiving peer for the stream or the connection.

HTTP/2 does not specify any algorithms, it defines only the format and the semantics of the `WINDOW_UPDATE` frame. HTTP/2 allows any implementation, leaving room for experimentation, that respect the following characteristics: flow-control is specific to the connection (hop by hop), based on `WINDOW_UPDATE` frames which is a credit-based scheme, directional and controlled by the receiver, and it cannot be disabled.

The initial flow-control window is 65.535 octets for streams until other value is announced by the receiver endpoint in the `SETTINGS_INITIAL_WINDOW_SIZE` parameter in a `SETTINGS` frame. The connection windows is set to the same value until a `WINDOW_UPDATE` frame is received for the connection.

Changes in the SETTINGS_INITIAL_WINDOW_SIZE can cause the flow-control window to become negative. Negative values are valid and senders must keep track of the flow-control window value and wait until it becomes positive to start sending flow controlled frames again. Senders must also be ready to receive data that exceeds the window limit prior to the processing of the SETTINGS frame by the remote endpoint.

Implementations can choose completely different algorithms based on what fits their requirements. Implementations are also responsible for managing how requests and responses are sent based on priority and dependencies to avoid head-of-line blocking, and managing the creation of new streams.

2.6.2 Stream Priority

The multiplexing nature of streams enable another feature in HTTP/2: stream priority. Clients can assign priority to streams, allowing the server to allocate resources based on its preference. This is very important when the resources are limited. This prioritization is not enforced, therefore only a suggestion.

The prioritization is notified by the client on the HEADERS frame for new streams or with a PRIORITY frame for already existing streams. Streams can be prioritized in two different ways: by making them dependant in other streams or by assigning them a weight relative to other streams with the same parent.

A stream can be given an explicit dependency on another stream. Streams that share parent dependency do not have any specific order, this is determined by the weight. The exclusive flag allows a stream to become the sole dependency of its parent, causing all child streams depend on this exclusive stream. A stream that is not dependent on any other stream is given a stream dependency of 0x0, which forms the root of the tree.

All dependent streams are allocated an integer weight between 1 and 256. Streams with the same parent should be allocated resources proportionally based on their weight, meaning that a bigger weight should guarantee more resources to be allocated.

If a stream is removed from the dependency tree, its dependent streams can be moved to become dependent on the parent of the closed stream. The weights are recalculated by distributing the weight of the closed stream proportionally based on the weight of it dependencies. Removed streams can cause a loss of some prioritization information. To avoid this problems, an endpoint should retain stream prioritization state for a period after streams become closed. This retention of information for streams are not counted toward the limit of streams set by SETTINGS_MAX_CONCURRENT_STREAMS.

By default, new streams are assigned a non-exclusive dependency on the stream `0x0` except for pushed streams, which are dependent on their associated stream. In both cases, streams are assigned a weight of 16.

2.7 Frames

The frame is the basic protocol unit. All frames are composed by a 9-octet header and a variable-length payload as shown in Figure 2.4. The payload depends completely on the type of frame sent.

The size of the frame payload is limited by the receiver, which announces it in the `SETTINGS` frame during the connection establishment. This value can be any value between 2^{14} (16.384) and 2^{24}-1 (16.777.215) octets, inclusive. All implementations must be able to handle at least 2^{14} bytes of data plus 9 bytes of headers.

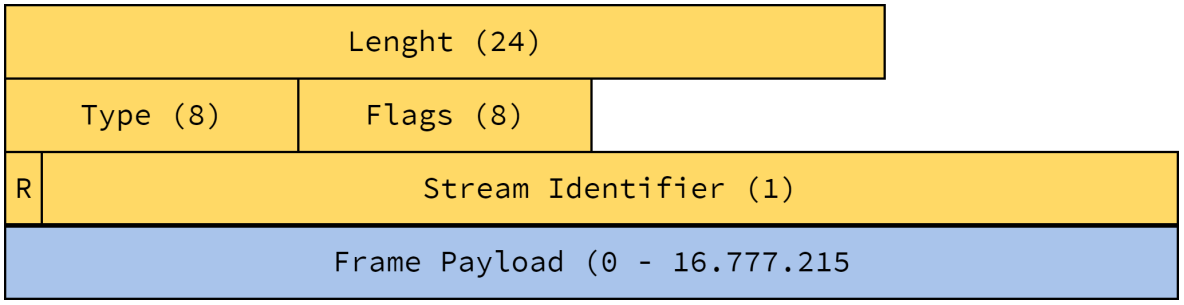

Figure 2.4: Frame layout

The header of an HTTP/2 frame is composed of the following fields:

- Length (24 bits): The length of the payload in octets, frame header length is not included.
- Type (8 bits): The type of the frame, determines its format and semantics.
- Flags (8 bits): Boolean flags related to the type of the frame.
- Reserved (1 bit): Reserved field. Nothing is defined for this bit, it must remain unset.
- Stream Identifier (31 bits): The stream identifier for the frame. The special value `0x0` is reserved for frames that are associated with the connection.

HTTP/2 specifies 10 different types of frames: `DATA`, `HEADERS`, `PRIORITY`, `RST_STREAM`, `SETTINGS`, `PUSH_PROMISE`, `PING`, `GOAWAY`, `WINDOW_UPDATE` and `CONTINUATION`. Each frame type can affect the connection as a whole or only a specified stream.

2.7.1 DATA

DATA frames (type=0x0) contains the data generated by the HTTP application. The length of these frames is variable, one or more DATA frames can be used to transport requests or responses payloads. DATA frames can contain random padding to obscure the size of the message to improve security.

These frames must be associated with a stream and are subject to flow control. They can be sent only if a stream is *open* or *half-closed (remote)*.

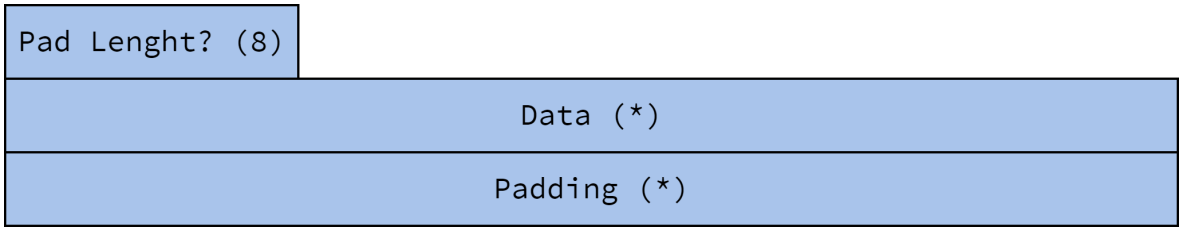

Figure 2.5: DATA frame payload

The DATA frame, Figure 2.5, has the following fields:

- Pad Length (8 bits): Length of the padding in octets. This field is optional and it is only present if the PADDED flag has been set.
- Data (variable): Contains the application data.
- Padding (variable): Padding octets with no semantic value. Must be set to zero.

DATA frames define the following flags in the header:

- END_STREAM (0x1): This flag indicates the end of the stream.
- PADDED (0x8): Indicates that the Pad Length field and any padding are present.

2.7.2 HEADERS

HEADERS frames (type=0x1) are used to open a stream and also to carry a header block segment. HEADERS frames are associated with a stream.

HEADERS frames change the state of the connection, which causes a block on it. If the header block does not fit in a HEADERS frame, one or more CONTINUATION frames must follow it with the rest of the header block. No other frames can be issued until the complete header block has been sent to the remote peer. To indicate the last frame containing a header block fragment the END_HEADERS flag must be set in frame header.

HEADERS frames can be sent on a stream in the *idle*, *open*, *reserved (local)* or *half-closed (remote)* state.

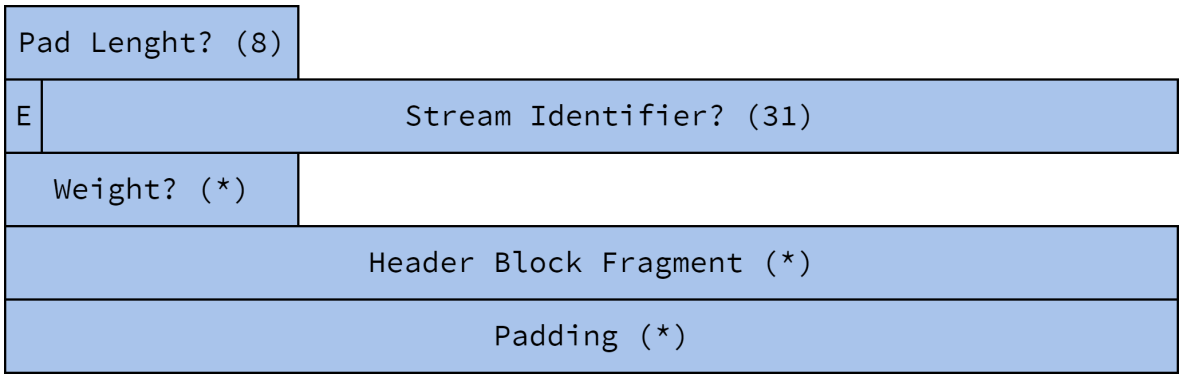

Figure 2.6: HEADERS frame payload

The HEADERS frame, Figure 2.6, contains the following fields:

- Pad Length (8 bits): Length of the padding. Only present if the PADDED flag is set.
- Exclusive (E) (1 bit): A flag indicating that the stream has a exclusive dependency. This field is only present if PRIORITY flag is set.
- Stream Dependency (31 bits): The stream identifier on which this stream depends. This field is only present if PRIORITY flag is set.
- Weight (8 bits): Integer representing a priority weight for the stream. One needs to be added for a value between 1 and 256. This is only present if PRIORITY flag is set.
- Header Block Fragment (variable): A header block fragment.
- Padding (variable): Padding octets.

The HEADERS frame define the following flags:

- END_STREAM (0x1): This flag indicates the end of the stream.
- END_HEADERS (0x4): Indicates the frame contains an entire header block and is not followed by any CONTINUATION frames.
- PADDED (0x8): Indicates Pad Length and Padding blocks are presents.
- PRIORITY (0x20): Indicates the presence of Exclusive Flag (E), Stream Dependency and Weight fields.

2.7.3 PRIORITY

The PRIORITY frame (type=0x2) specifies the sender-advised priority of a stream. PRIORITY stream is a subset of the HEADERS frame, it only includes the blocks related to priority for streams.

PRIORITY frames can be sent for any stream in *idle* or *closed* state, in order to allow changes in the prioritization of a group dependent on the stream.

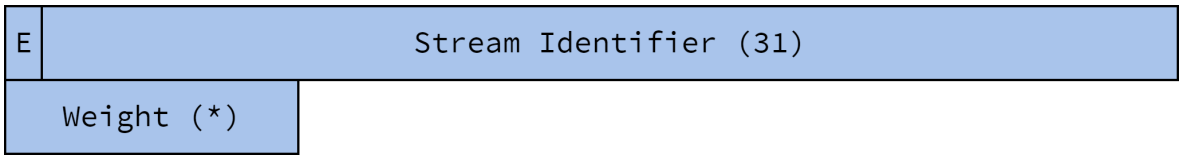

Figure 2.7: PRIORITY frame payload

The PRIORITY frame, Figure 2.7, contains the following fields:

- Exclusive (E) (1 bit): A flag indicating that the stream has a exclusive dependency.
- Stream Dependency (31 bits): A stream identifier for which this stream depends on.
- Weight (8 bits): An integer that represents a priority weight for the stream. One needs to be added to obtain a weight value between 1 and 256.

For PRIORITY frames no flags are defined.

2.7.4 RST_STREAM

The RST_STREAM frame (type=0x3) indicates the termination of a stream. This frame has two use cases: to request a cancellation of a stream or to indicate an error occurred.

RST_STREAM cannot be sent for a stream in an idle state. This frame will cause to the stream to enter into the *closed* state. Receivers should not send more data in that stream, the sender of the frame must be able to receive frames sent by the other endpoint prior to receive it.

Figure 2.8: RST_STREAM frame payload

RST_STREAM frame, Figure 2.8, contains a single block (32 bits) indicating the error occurred.

There is no flags defined for RST_STREAM frame.

2.7.5 SETTINGS

The SETTINGS frame (type=0x4) carries configuration parameters to define the communication between the endpoints. This frame describes the characteristics of the sender. These parameters are not negotiated, each peer can have a different configuration. SETTINGS frames must be acknowledged by the recipient.

SETTINGS must be sent at the beginning of the connection by both peers, and also at any point during the connection. The settings parameters must be kept by the receipts with the last value received. Parameters apply to the connection, not to individual streams.

Synchronization is very important to keep the flow-control. All the parameters in the SETTINGS frame must be processed in the order of appearance. When all the value have been processed, the recipient endpoint must send a SETTINGS frame with the ACK flag set to acknowledge this. If no ACK is received in a time manner, the sender may issue a connection error of type SETTINGS_TIMEOUT.

There is only one flag defined for the SETTINGS frame:

- ACK (0x1): Indicates the SETTINGS frame has been received and applied to the connection. No other information should be send, the payload must be empty.

The payload of a SETTINGS frame is composed by zero or more parameters blocks.

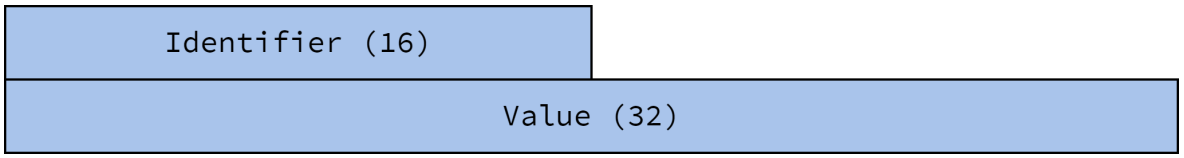

Figure 2.9: SETTINGS frame payload

Parameter blocks, Figure 2.9, contains two fields:

- Identifier (16 bits): The identifier of the parameter.
- Value (32 bits): The value for the setting specified by the sender.

The parameters defined are:

- SETTINGS_HEADER_TABLE_SIZE (0x1): Indicates the maximum size of the header compression table used in the connection to decode headers blocks, in octets. The initial value is 4.096 octets.
- SETTINGS_ENABLE_PUSH (0x2): This boolean parameter indicates if server push is permitted or disabled. Servers must not send PUSH_PROMISE frames if this parameter is disabled. The initial value is 1, server push enabled.
- SETTINGS_MAX_CONCURRENT_STREAMS (0x3): Indicates the maximum number of streams that can live concurrently in the connection. There is no initial limit for it, but this value is recommended to do not be smaller than 100. The value can be set to 0 in order to avoid the creation of new streams.
- SETTINGS_INITIAL_WINDOW_SIZE (0x4): This setting indicates the initial window size in octets for the sender, it applies to all streams. The initial value is 2^{16}-1 (65.535) octets, and the maximum value is 2^{31}-1.

- `SETTINGS_MAX_FRAME_SIZE` (`0x5`): Indicates the size of the of the largest payload that the sender can handle, in octets. The initial value is 2^{14} (16.384) octets, which it is the minimum value that all the implementations must handle. The maximum value is the maximum length for a frame: 2^{24}-1 (16.777.215) octets.
- `SETTINGS_MAX_HEADER_LIST_SIZE` (`0x6`): This parameter indicates the maximum size of the header list that the sender can accept, in octets. This value is based on the uncompressed size of the headers. including the length of the name and value in octets plus an overhead of 32 octets for each header field. The initial value of this parameter is unlimited.

Unknown or unsupported identifiers must be ignored by the receiver endpoint.

2.7.6 PUSH_PROMISE

The `PUSH_PROMISE` frame (type=`0x5`) is used to advertise the other endpoint in advance about the streams the sender is going to initiate. Along with the reserved stream identifier, a `PUSH_PROMISE` frame includes a header block with the information about the resource to send. `PUSH_PROMISE` frames can be sent only if `SETTINGS_ENABLE_PUSH` parameter is set to true by the receiving peer.

`PUSH_PROMISE` frames contain a header block, and like `HEADERS` frames, they are blocking. If a header block does not fit in a single `PUSH_PROMISE` frame, one or more `CONTINUATION` frames must be sent after until the whole header block is sent. No other frames in any stream can be sent until `END_HEADERS` flag is received.

Receiving endpoints can cancel promised streams by sending a `RST_STREAM` with the promised stream identifier back to the sender. A receiver needs to be able to handle more `PUSH_PROMISE` frames created by the sender before `RST_STREAM` has been processed.

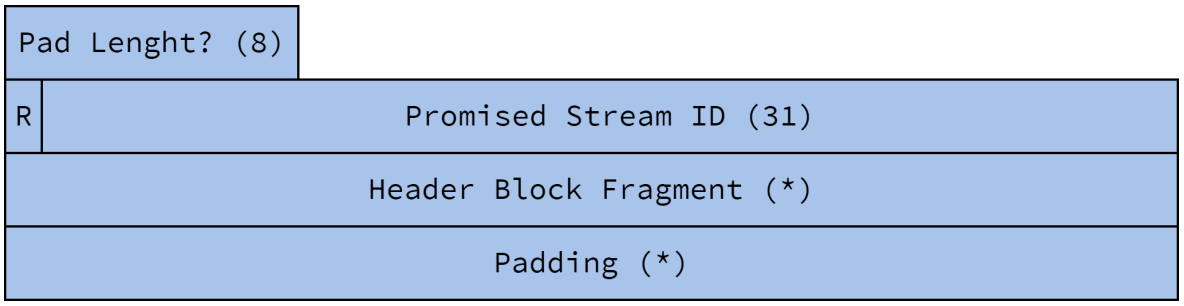

Figure 2.10: `PUSH_PROMISE` frame payload

The `PUSH_PROMISE` frame, Figure 2.10, is composed by the following fields:

- Pad Length (8 bits): Length of the padding in octets.
- R (1 bit): Single reserved bit.
- Promised Stream (31 bits): A stream identifier reserved by the sender.
- Header Block Fragment (variable): A header block fragment.
- Padding (variable): Padding octets with no semantic value.

`PUSH_PROMISE` frames define the following flags:

- `END_HEADERS` (`0x4`): Indicates the frame contains an entire header block and is not followed by any `CONTINUATION` frames.
- `PADDED` (`0x8`): Indicates Pad Length and Padding blocks are present.

2.7.7 PING

The `PING` frame (type=`0x6`) is used to calculate the minimal round-trip and to determine if the connection is still functional, the stream identifier must be set to `0x0`, as it refers to the connection.

Receivers must acknowledge a `PING` frame by sending another `PING` frame with the same opaque data and the `ACK` flag set. This response should take priority over other frames.

Figure 2.11: `PING` frame payload

The payload of a `PING` frame, Figure 2.11, is 8 octets of opaque data, any value is valid.

The `PING` frame only defines one flag:

- `ACK` (`0x1`): This flag indicates the frame is a response to a previous `PING` frame.

2.7.8 GOAWAY

The `GOAWAY` frame (type=`0x7`) is used to finish the connection, either with a gracefully shutdown or due to an unrecoverable error. Due to race conditions between when the frame is sent and until the peer processes it, the `GOAWAY` frame includes the identifier of the last processed or might be processed stream.

Once GOAWAY frame is sent, the sender can discard frames with a higher stream identifier, and receivers must not open more streams in the connection. Not all frames should be discarded, frames that alter the connection state like HEADERS, PUSH_PROMISE, and CONTINUATION must be processed to ensure the header block is consistent. Also DATA frames must be counted toward the flow-control window. Unprocessed streams can be retried again by the peer in a new connection.

GOAWAY frame should be sent prior to close the connection, so receivers can know which streams have been processed. Receivers should also send a GOAWAY frame before terminating the connection. The sender of the GOAWAY frame can maintain the connection open until all in-process streams are complete in the case of gracefully shutdown.

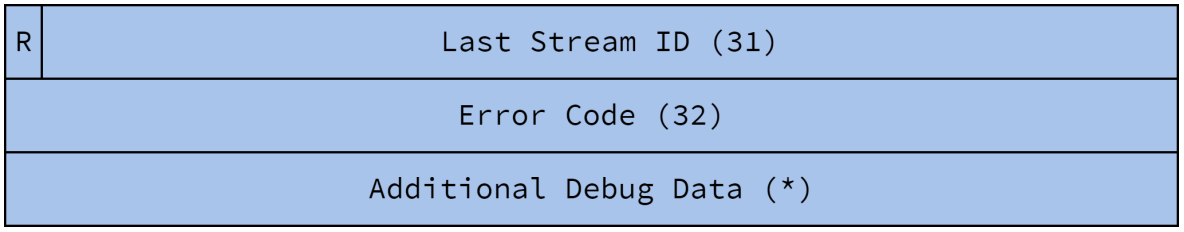

Figure 2.12: GOAWAY frame payload

The GOAWAY frame, Figure 2.12, defines the following blocks:

- R (1 bit): Single reserved bit.
- Last Stream (31 bits): Last stream identifier processed.
- Error Code (32 bits): Reason for closing the connection.
- Debug Data (variable): Optional field to include debug data for diagnostic purposes.

GOAWAY frames do not define any flags.

2.7.9 WINDOW_UPDATE

The WINDOW_UPDATE frame (type=0x8) is used to implement flow control on both levels: on individual streams and on the entire connection.

Only DATA frames are subject to flow control. Other frames are exempt and must be accepted and processed by the receiving endpoint. If the peer is unable to handle the frame, a FLOW_CONTROL_ERROR may be sent as a response within the stream or for the whole connection. All flow-controlled frames must be counted by the receiving endpoint, even if the frame is in error or the peer has requested a cancellation. This is necessary to keep a consistent value in both endpoints.

The WINDOW_UPDATE frame, Figure 2.13, defines two fields:

Figure 2.13: WINDOW_UPDATE frame payload

- R (1 bit): Reserved bit, currently with no use.
- Window Size Increment (31 bits): Number of octets that the sender can transmit, the value for the window increment must be between 1 and 2^{31}-1 (2.147.483.647) octets.

No flags are defined for WINDOW_UPDATE frames.

2.7.10 CONTINUATION

The CONTINUATION frame (type=0x9) contains header blocks fragments. If the data of the HEADERS and PUSH_PROMISES cannot fit in the payload, one or more CONTINUATION frames must be sent with the remaining information until the flag END_HEADERS is present.

CONTINUATION frames are blocking and no other frame can be sent in the stream after a HEADERS or a PUSH_PROMISE until the header block has been sent as they are stateful.

Figure 2.14: CONTINUATION frame payload

CONTINUATION frames, Figure 2.14, contains a header block fragment in the payload.

The CONTINUATION frame only defines one flag:

- END_HEADERS (0x4): This flag indicates the end of headers.

2.8 Error Handling

There are two classes of errors in HTTP/2: errors conditions related to the entire connection, which becomes no longer usable, and errors related to individual streams.

A connection error make the current connection unusable for further communication, which might be for example due to a corruption of the state or a violation of the protocol. Endpoints that encounters themselves in such situation should send a GOAWAY frame indicating the last stream successfully processed along with the error code associated.

After sending the frame, the connection must be closed. Streams in an *open* or *half-closed* state when the connection is closed, cannot be automatically retried.

GOAWAY frames are not acknowledge, hence, not reliable. This works on a best-effort attempt to communicate with the other peer.

On the other hand, errors related to specific streams do not affect the entire connection nor other streams within it. RST_STREAM frames are used to communicate to the remote endpoint the error occurred. Due to the asynchronous communication, senders must be prepared to receive any frames sent prior to the reception of this frame by the remote peer. These frames can be discarded unless they change the connection state.

RST_STREAM frames should not be sent more than once for the same stream, but if the receiver keep sending frames on that stream for a time longer than a round-trip, more frames can be issued by the sender. RST_STREAM frames are not acknowledge, no other RST_STREAM frame should be sent to confirm the reception of a RST_STREAM frame.

2.8.1 Error Codes

Error codes are 32-bits fields used in RST_STREAM and GOAWAY frames to indicate the reason for the error. Some error codes only apply to either streams or the connection, but they share a common space code.

The following codes are defined in the protocol:

- NO_ERROR (0x0): This code is associated with a non error condition. Might be used in GOAWAY frame to indicate a graceful shutdown.
- PROTOCOL_ERROR (0x1): The endpoint detected an unspecified error, only used when a more specific code is not available. For example, when a peer tries to open more concurrent streams that it is allowed by the settings or when a frame that must specify an individual stream uses the special value 0x0 reserved for the connection.
- INTERNAL_ERROR (0x2): The endpoint found an unexpected internal error.
- FLOW_CONTROL_ERROR (0x3): Endpoints return this code if the flow-control has not be respected by the sending peer in any of the windows, stream or connection.
- SETTINGS_TIMEOUT (0x4): If a SETTINGS frame has not been acknowledged in a manner time, the sender may issue a connection error with this code.
- STREAM_CLOSED (0x5): Used when a frame is received in a *half-closed* stream.
- FRAME_SIZE_ERROR (0x6): If the frame does not match the value specified in the length field. For fixed length frames, if the length does not match the expected value. In the case of the SETTINGS frame, a multiple of 6 is expected.

- `REFUSED_STREAM` (`0x7`): If the stream is refused by the endpoint prior to any application processing. One of the use cases is to discard a reserved stream announced by a `PUSH_PROMISE` frame.
- `CANCEL` (`0x8`): This code is used when the endpoint does not longer require the stream. Mostly used to cancel streams reserved by a `PUSH_PROMISE`.
- `COMPRESSION_ERROR` (`0x9`): If the endpoint is unable to maintain the header compression for the connection.
- `CONNECT_ERROR` (`0xA`): The connection established in responses a `CONNECT` request was reset or abnormally closed.
- `ENHANCE_YOUR_CALM` (`0xB`): If an endpoint depends that the peer is generating too much load and causing problems, this code should be sent by the endpoint.
- `INADEQUATE_SECURITY` (`0xC`): This code is used if the underlying transport does not meet the minimum security requirements.
- `HTTP_1_1_REQUITED` (`0xD`): The endpoint requires that HTTP/1.1 be used instead of HTTP/2.

Error codes not included on the list or unsupported codes must not trigger any special behavior, they may be treated as `INTERNAL_ERROR`.

2.9 Server Push

This is one of the new features of HTTP/2, that allows servers to send multiple responses to a client for a single request. Clients need to process the data before requesting new resources, this gives the server the ability to get ahead and send those resources before the client requests them, eliminating the extra latency.

Promised requests must be cacheable, must use a safe method and must not include a request body. Pushed responses that are cacheable can be stored by the client if it implements an HTTP cache.

Server push is semantically equivalent to a server responding to a request, but in this case the server sends the request as a `PUSH_PROMISE` frame. `PUSH_PROMISE` frames include a header block containing the complete set of headers fields. Just like `HEADERS` frames, if the request does not fit in a single frame, one or more `CONTINUATION` frames must be used until the header block is sent. Push responses can be sent only to requests with no request body.

Push responses must be associated to the request that originated them. There responses are sent in their own stream, that is announced in the `PUSH_PROMISE` frame. Servers

should send PUSH_PROMISE frames prior to sending any DATA frame that reference the promised responses to avoid race with the client requests.

Once the PUSH_PROMISE frame has been sent, the server can start pushing the response in the new stream. This response starts with a HEADERS frame to prepare the client before send the DATA frames until END_STREAM flag is set.

Clients can refuse the promised response, but once it has been accepted, clients should not issue any requests for the promised responses. In order to cancel or refuse a pushed response, clients can send a RST_STREAM frame with either CANCEL or REFUSED_STREAM codes.

This feature, while available in the protocol, can be disabled by clients by sending a 0 in the SETTINGS_ENABLE_PUSH parameter in the SETTINGS frame.

Only servers have the ability to push resources, any attempt by a client to send a PUSH_PROMISE frame must be treated by the server as a connection error of the type PROTOCOL_ERROR. The same applies to clients that must reject any attempt by the server to change the SETTINGS_ENABLE_PUSH settings value to anything other than 0.

Chapter 3

Software Support

Since the HTTPbis working group started working on the new version of HTTP, the community has been following the specification drafts very closely and developing libraries in all notable programming languages.

Even before the publication of the HTTP/2 standard[1], most of the major browsers, such as Chrome and Firefox have been able to support it[9][10], accommodating the deployability of the protocol in the real Internet.

3.1 Browsers

Support for the new protocol is already in widespread use across modern browsers, increasing its adoption with the new releases.

As explained before, HTTP/2 can be used in plain TCP or via TLS. Major browsers like Chrome[11] and Firefox[11][12] have announced they will only implement HTTP/2 over TLS, enforcing communications to be always encrypted. On the other side, Internet Explorer will support both types of communication: plain TCP and secure[11].

Google Chrome have been rolling out support for HTTP/2 since version 40[9]. Chrome also supports SPDY since version 6.

First implementation of HTTP/2 in Mozilla Firefox were in version 35[10]. In version version 36 the support for the official final "h2" protocol for negotiation was added. Along with it, the support for the drafts IDS -14 and -15. The current version of Firefox, 38, implements the draft ID-16, along with drafts ID-14 and ID-15. It also includes the IETF drafts for opportunistic security over h2, via the Alternate-service mechanism.

Internet Explorer 11 supports HTTP/2 but only on Windows 8 in a Tech Preview[13]. Microsoft Edge, the new browser developed by Microsoft to substitute Internet Explorer, will come with support for HTTP/2 from the beginning[14].

Safari will support HTTP/2 in the version 9. Safari 9 will be released at the same time as the new operative system of Apple iOS 9 in September 2015[15].

3.2 Web servers

In contrast with the incredible work and effort done for supporting HTTP/2 in browsers, the implementation in general purpose Web servers has not been alike.

Right now, none of the three most used Web servers[16], Figure 3.1, have full support for it neither is expected before the end of the year.

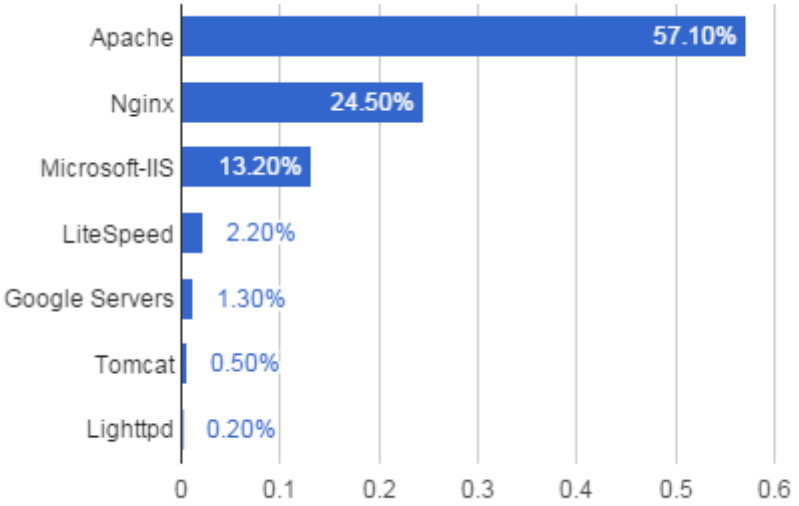

Figure 3.1: Web servers usage by W3Techs.com

Apache httpd does not have plans to add support for HTTP/2 in the short term[17]. There is an experimental module created by a third-party developer to support HTTP/2, mod_h2[17], based on nghttp2 library.

NGINX Inc., on the other hand, plans to releases versions of both of their software, NGINX and NGINX Plus, by the end of 2015 with support for HTTP/2[18].

IIS is adding HTTP/2 support in its version 10, which it is included in Windows Server 2016. The release date of Windows Server 2016 is expected to be in early 2016[19].

LiteSpeed, the next server in the W3Tech list, fully supports HTTP/2 and it is production ready[20]. LiteSpeed is the proprietary version of the open-source project OpenLiteSpeed developed by LiteSpeed Technologies.

Another production ready server is H2O developed by Kazuho Oku, it has implemented HTTP/2 since version 1.3.0[21].

In Java world, both Jetty[22] and Netty have been fast implement it, those libraries are production ready since the last draft. There are restrictions to this implementation as there is no ALPN support in the current Java version, Oracle Java 8, it will be included in the next release. Until that support is ready, Jetty developers created a library to add support for ALPN into OpenJDK 7 and 9[23].

There is already in almost every programming language a HTTP/2 server implemented natively, mostly open source projects. While this implementations are functional, they might not be production ready yet as unknown bug can be present due to the short-lived time of the projects.

Daniel Stenberg, author and maintainer of cURL and libcurl, estimates that by the end of 2015 the leading HTTP server products -with a share of more than 80% of the server market- will support HTTP/2[24].

3.3 Proxies, Caches and CDNs

Not only clients and servers need to support HTTP/2. There are other elements in the network that act as intermediaries, providing extra functionality or features, like proxies, caches or CDNs.

For example, content delivery networks help to spread the load between all the Internet and also place the resources closer to the end user to reduce latency.

Reverser proxy caches are very used in high demanded websites. They cache the results provided by the different back-ends to save time, resources and energy by avoiding it to generate the same content thousands of times.

HAProxy, one of the most high performance TCP/HTTP load balancer, developers have been preparing themselves with a internal refactor in their current version 1.6 and plan to roll out support for HTTP/2 by the end of the year, in the release of its next minor version 1.7[25].

Squid is probably the most famous caching and forwarding web proxy. Their developers are working on the implementation of HTTP/2 for its version 4, which it is in development at the moment. There is no fix date as it is a open source project and depends mostly on available developers time, but they are aiming for the beginning of 2016[26].

Apache Traffic Server, another important caching proxy, has experimental support for HTTP/2 based on the draft ID-16[27].

Varnish, one of the most performant caching HTTP reserve proxy -they define themselves as an HTTP accelerator-, will support HTTP/2 through the PROXY protocol[28]. Vanish uses Hitch, a network proxy for TLS/SSL termination. Hitch plans to have support in the first quarter of 2016.

One of the most interesting proxy servers is nghttpx[29], a proxy translation protocol between HTTP/2 and other protocols (e.g. HTTP/1.1, SPDY) build on top of the library nghttp which supports h2 and h2c. The fact of being ready for production, and the use of nghttp, seems very likely that some companies will use it as entry point standing before other web servers until they have support for HTTP/2.

Within the Content Delivery Networks, there is not much news at the moment. Akamai software currently includes beta support for it, their main goal is to have fully support before the end of 2015[30]. CloudFlare will support HTTP/2 once NGINX releases support for it. Amazon has not made any public announcement.

3.4 Testing tools

Each protocol, especially if it is binary, requires a big effort by the community of developers to implement it in their software or create new software around it.

Testing tools help us to understand in details what is in the network and how our applications are using with the protocol.

The most well-know tool with which to understand and inspect any kind of network traffic is Wireshark. It is in fact, the de facto software used as standard across many industries and educational institutions.

Wireshark has been following closely the standardization of HTTP/2, adding it into their supported protocols since 2013. Wireshark has already fully support for HTTP/2 since release 1.12[31].

Other testing tools that can help to understand the right direction of HTTP/2 come directly with the browsers. They are able to debug and inspect functionality to help developers, like Network Monitor[32] in Mozilla Firefox and DevTools[33] in Google Chrome.

They also exists as plugins like the open-source software Firebug. Also commercial software for inspecting specialized in web performance is becoming available to the developers. A basic edition with no charge is HttpWatch [34].

H2i is an interactive HTTP/2 console debugger, currently under development, in order to provide a "telnet" feel functionality[35]. This is part of a work-in-progress HTTP/2 server and client written in Go.

3.5 Others

The command line tool curl has support for HTTP/2 since version 7.36, it was added into the libcurl library using nghttp2. Curl supports HTTP/2 over TLS and plain TCP connection[36].

GNU Wget does not support HTTP/2 at the moment, neither there is a plan to adopt it in the short term.

One of the most important implementations is nghttp2, an HTTP/2 library written in C. Used in the Apache httpd module mod_h2 and in other programming languages thought bindings. It includes the command line client nghttp[37], with support for all types of connection: secure, HTTP Upgrade and plain text.

Many other client implementations become available during the last months: node-http2[38] for JavaScript, http2-perl[39] for Perl, http2[40] for Go, Hyper[41] for Python, OkHttp[2] for Java, http-2[42] for Ruby, etc.

Chapter 4

HTTP/2 Deployment

In this chapter we try to understand the HTTP/2 adoption in today's Internet. The new protocol was approved by the IESG in February 2015 and published as an RFC in May of the same year, making Web companies able to start rolling out HTTP/2 web servers into their production environments.

HTTP/2 is intended to be the future of the Web and how widespread the implementation of the protocol has been since the standard was completed is one of the main questions which can help determine if the Web is on board.

To answer the question, we build a measurement tool to shed some light on the implementation of HTTP/2 across the Internet. The tool runs on Cloud9 and crawls a list of Web sites making requests to look for servers that implement the protocol.

For each Web site, the tool checks if HTTP/2 is supported over TLS, via HTTP Upgrade mechanism and also, if the server responds to a direct HTTP/2 request over plain TCP.

As a significant sample of the Internet, we use the top 1 million websites ranked by Alexa for our test. An updated free list is available at the following URL:

http://s3.amazonaws.com/alexa-static/top-1m.csv.zip

The sample used for the measurement dates from the 20th of June 2015.

Based on the information in Chapter 3, we can anticipate, as the most used web servers Apache, NGINX and IIS (about 95% usage, see Figure 3.1) do not support HTTP/2, consequently the vast majority of the sites in the top 1 million of Alexa do not implement HTTP/2 yet.

4.1 Measurement Platform

We decide to use the library nghttp2. We use the command line tool included in the library, overcoming the complexity of the language C, `nghttp`.

To collect all the information for the measurement, we create a bash script on top of nghttp to save the results and also to add extra functionality in order to obtain reliable results.

The script processes every Alexa URL making requests to check HTTP/2 support using 3 different methods: over TLS, using HTTP Upgrade and, finally, making a direct request over plain TCP. The script reads the CSV file downloaded from Alexa in its standard input and outputs the results into another CSV file.

While nghttp is a very complete tool, it has a limitation for this implementation: it does not behave like browsers. It just performs a request to a specified URL and shows all the information related to the connection, however HTTP redirects are not obeyed.

This is very important because the list offered by Alexa only includes the domain root and the top level domain extension. In most cases, the Web sites are hosted under sub domains, typically under www.

To obtain all the necessary information, the script performs the following logic for each Web site in the list:

- It makes a request to the Web site URL prefixed by www.
- If an error occurs, it makes a request without the www prefix.
- If the request returns a 3XX status (redirection), it makes another request to the new location until a non 3XX status is received.
- If the redirection is to a different scheme, the script does not follow the redirection, as this is checked in the other methods.
- If a web supports HTTP/2, then the tool saves the last valid request, the status code of the response, the scheme of the location header if it exists and the server. If it does not, then the tool records the failure.

Web sites like blogspot, support HTTP/2 over TLS but do not serve any content, instead they respond with a redirection to their non-secure version. Over plain TCP they do not support HTTP/2, not even using HTTP Upgrade. It is very important to save the HTTP status of the last page requested to better understand the implementation of HTTP/2. Not only do we want to know if the server supports HTTP/2, we also want to know if the server provides any content using the new protocol.

Attempting to perform HTTP/2 requests to those Web sites, nghttp could encounter a series of errors. First, not all websites support HTTP and HTTPS. Attempting to connect on port 80 or 443 can be refused by the servers. And secondly, the list of URLs contains the top ranked Web sites globally, some of which are hosted in China and are unavailable from outside of their frontiers.

Our tool is able to recognize these errors and it lists them as follows:

- *No HTTP/2 response*
  There was no HTTP/2 response and the connection was closed by the server.
- *No HTTP/2 negotiated (only applies to secure connections)*
  No support is shown for HTTP/2 during the TLS handshake (only for servers that support ALPN and/or NPN).
- *Unable to resolve the domain (DNS)*
  The script is unable to resolve the host name to an IP address.
- *Timeout (defined as 3 seconds)*
  There is no response to the requested URL, but the connection is not refused neither closed by the server.
- *Connection refused*
  The server refuses the client request.
- *Unknown*
  The response was not processed and no errors are shown by `nghttp`.

The script runs in a cloud platform concurrently, 5 scripts running at the same time, and once it is completed, the resulting CSV files are merged and downloaded from the cloud.

To run the scripts, we use the platform Cloud9. It offers a free plan with 512MB of RAM and 1GB of disk space. To collect all the information for the list of 1 million of websites provided by Alexa, it takes approximately 1 week on their servers.

The CSV file with all the data is imported into a SQLite database for the post-processing phase.

4.2 Analysis and results

We collect data from the 24th of August to the 30th of the same month.

In this section, we present the results about the support of HTTP/2 over TLS, then we show the results about HTTP/2 using the Upgrade mechanism and over plain TCP.

Table 4.1 shows an overview of the results across the 3 methods.

Support indicates the number of servers that replied with an HTTP/2 response. The *content* column shows the number of Web sites whose content is actually served over HTTP/2.

The *Top pages* column indicates the number of Web sites tested. They are evaluated in order based on the ranking provided by Alexa. The values are selected in a way that each order of magnitude in a logarithmic chart will contain 4 marks.

	Over TLS		Upgrade		Plain TCP	
Top pages	**Support**	**Content**	**Support**	**Content**	**Support**	**Content**
10	3	3	0	0	0	0
20	5	5	0	0	0	0
35	11	9	0	0	0	0
60	18	14	0	0	0	0
100	24	19	0	0	0	0
200	39	32	0	0	0	0
350	56	48	0	0	0	0
600	69	57	0	0	0	0
1 000	83	67	0	0	0	0
2 000	102	78	0	0	0	0
3 500	122	92	0	0	0	0
6 000	146	113	0	0	0	0
10 000	173	133	0	0	0	0
20 000	254	204	0	0	0	0
35 000	369	300	0	0	0	0
60 000	560	442	0	0	0	0
100 000	947	716	0	0	0	0
200 000	2 339	1 512	1	0	3	0
350 000	5 113	2 954	2	1	6	1
600 000	10 919	5 576	5	3	11	4
1 000 000	22 653	10 162	16	10	26	14

Table 4.1: Implementation of HTTP/2

The next sections explain the results in detail, considering the three cases separately.

4.2.1 HTTP/2 Over TLS

Table 4.2 shows the results for the HTTP/2 requests over TLS.

For a better understanding, we include an extra group of columns with the percentage of the total. The last column (label *% Content*) indicates the percentage between

the number of servers that return content over HTTP/2 and the ones that support the protocol.

	Numbers		Percentage		
Top pages	**Support**	**Content**	**Support**	**Content**	**% Content**
10	3	3	30.00%	30.00%	100.00%
20	5	5	25.00%	25.00%	100.00%
35	11	9	31.43%	25.71%	81.82%
60	18	14	30.00%	23.33%	77.78%
100	24	19	24.00%	19.00%	79.17%
200	39	32	19.50%	16.00%	82.05%
350	56	48	16.00%	13.71%	85.71%
600	69	57	11.50%	9.50%	82.61%
1 000	83	67	8.30%	6.70%	80.72%
2 000	102	78	5.10%	3.90%	76.47%
3 500	122	92	3.49%	2.63%	75.41%
6 000	146	113	2.43%	1.88%	77.40%
10 000	173	133	1.73%	1.33%	76.88%
20 000	254	204	1.27%	1.02%	80.31%
35 000	369	300	1.05%	0.86%	81.30%
60 000	560	442	0.93%	0.74%	78.93%
100 000	947	716	0.95%	0.72%	75.61%
200 000	2 339	1 512	1.17%	0.76%	64.64%
350 000	5 113	2 954	1.46%	0.84%	57.77%
600 000	10 919	5 576	1.82%	0.93%	51.07%
1 000 000	22 653	10 162	2.27%	1.02%	44.86%

Table 4.2: Implementation of HTTP/2 over TLS

Figures 4.1 and 4.2 show the data of the table in a logarithmic representation and a percentage representation.

The results show that the adoption of the protocol by the top 100 is very high, a 24% of support and 19% of content. On the other hand, the adoption across the top 1 million is only the 2.27% of support and 1.02% of content.

This is an expected value as the big players of the Internet are the most interested in adopting HTTP/2, offering better services and user experience, and also in improving the usage of their massive infrastructures.

Table 4.3 shows for each Web server the number of Web sites hosted. Label *N* indicates the number of Web sites, and the same number is reported in percentage under the column %.

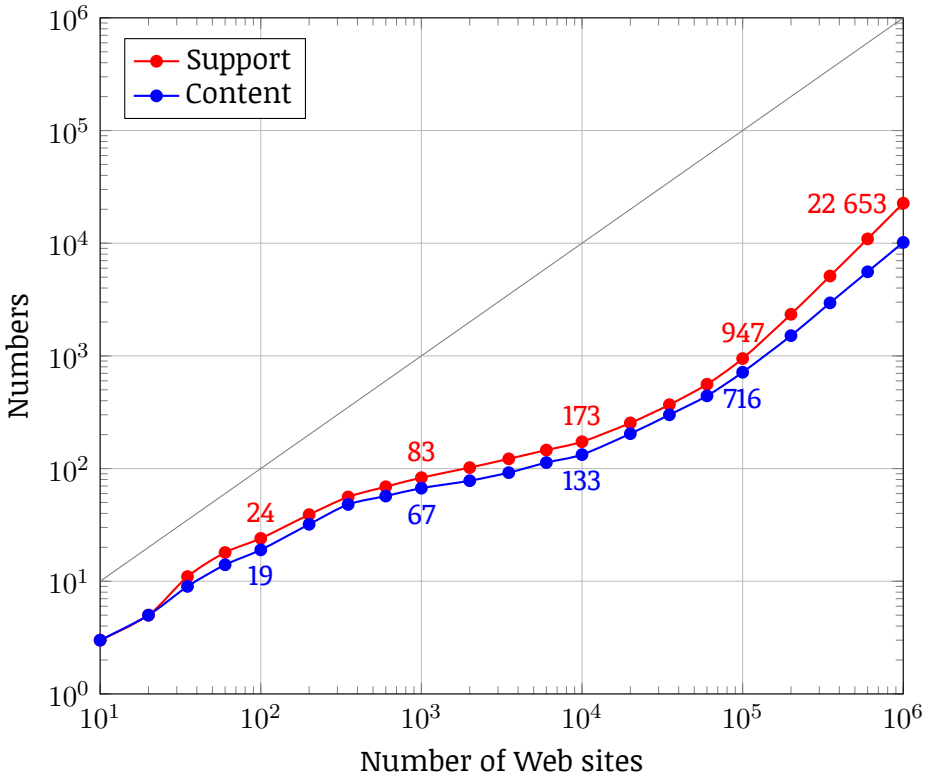

Figure 4.1: Logarithmic representation of HTTP/2 support over TLS

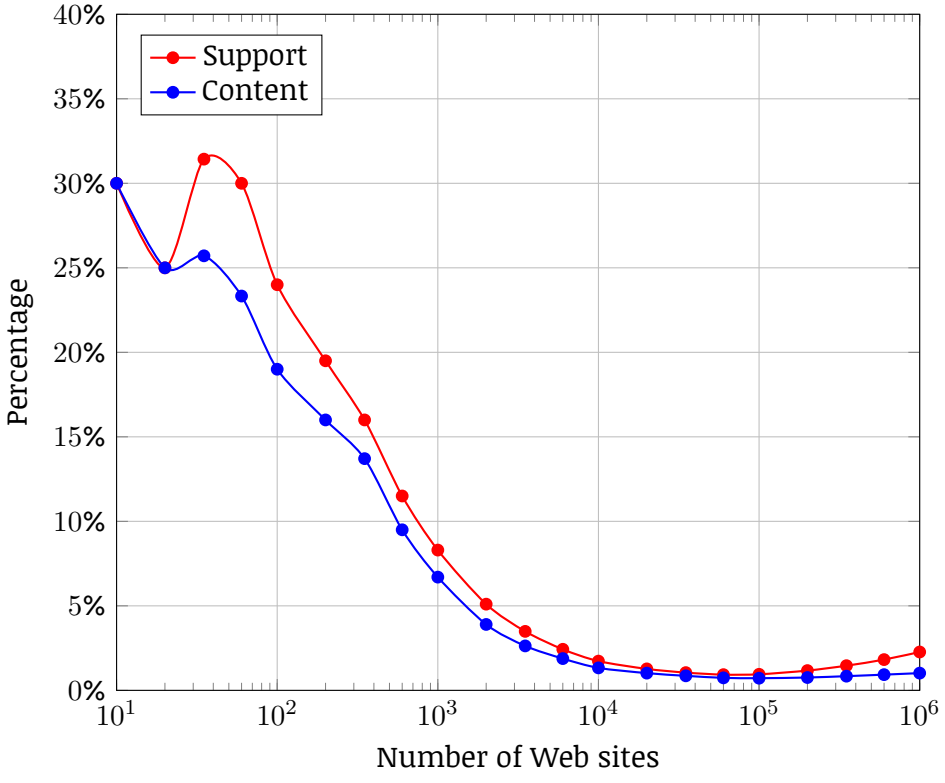

Figure 4.2: Percentage representation of HTTP/2 support over TLS

Server	N	%
Google	12 771	56.45%
LiteSpeed	9 823	43.42%
Apache	19	0.08%
H2O	8	0.03%
nghttpx	3	0.01%
Other	30	0.13%
Total	12 491	100.00%

Table 4.3: Servers with support for HTTP/2 over TLS

This results match the information of Section 3.2, LiteSpeed as the only general purpose Web server production ready for HTTP/2 and Google servers, as one of the biggest players on the Internet, with their own private software.

The list provided by Alexa does not deduplicate domains with different top-level domains. The most famous search engine around the world, Google, appears 197 times on the list, 105 of them in the top 10,000.

Table 4.4 shows the number of Google main domains compared with the total of results.

One single Web site with multiple different top-level domains powered by their HTTP/2 ready servers has big impact in the results. In the top 1,000 almost 76% of the Web sites that support HTTP/2 are the main domain of Google, and 94% of the ones that provide content. In the top 10.000 those percentages are still high, with a 60% of support and 79% of content.

After the top 100,000, the number of sites that serve content over HTTP/2 versus the servers that support it drop from over 75% to about 50%. Having a deeper look on the data, 11,222 out of 12,491 (89.84%) of the websites that do not return content over TLS are hosted by the Google blogging platform Blogger, under a blogspot domain.

Google servers reply with a 301 HTTP/2 response redirecting to a non secure URL. Note that the major browsers[11] only support HTTP/2 over TLS. Even if the sever supports it over plain TCP, most of the users will not benefit from it.

Table 4.5 resumes the reasons why no HTTP/2 content is served.

1,269 sites that do not server content in HTTP/2, 544 return a 3XX and 597 return a 4XX status code, both with an http URL in the location header.

For the last 128 Web sites left, multiple are the reasons to do not server content over HTTP/2: for example, GMail and other Google services redirect the client to a specific

		Support		Content	
Top pages	**Google**	**All**	**%**	**All**	**%**
10	1	3	33.33%	3	33.33%
20	3	5	60.00%	5	60.00%
35	7	11	63.64%	9	77.78%
60	12	18	66.67%	14	85.71%
100	17	24	70.83%	19	89.47%
200	29	39	74.36%	32	90.63%
350	45	56	80.36%	48	93.75%
600	54	69	78.26%	57	94.74%
1 000	63	83	75.90%	67	94.03%
2 000	71	102	69.61%	78	91.03%
3 500	83	122	68.03%	92	90.22%
6 000	98	146	67.12%	113	86.73%
10 000	105	173	60.69%	133	78.95%
20 000	124	254	48.82%	204	60.78%
35 000	135	369	36.59%	300	45.00%
60 000	144	560	25.71%	442	32.58%
100 000	155	947	16.37%	716	21.65%
200 000	169	2 339	7.23%	1 512	11.18%
350 000	181	5 113	3.54%	2 954	6.13%
600 000	191	10 919	1.75%	5 576	3.43%
1 000 000	197	22 653	0.87%	10 162	1.94%

Table 4.4: Comparative of Google versus total implementation of HTTP/2

Reason	**N**	**%**
Blogger/blogpsot	11 222	89.84%
http + 3XX status	544	4.36%
http + 4XX status	597	4.78%
Unknown	128	1.02%
Total	12491	100.00%

Table 4.5: HTTP/2 Web sites with no content returned

page to authenticate yourself before delivering content. Others do not have a valid certificate.

A part of those 128 Web sites return content over HTTP/2, but the number is very small compared with the sample that we consider it as false negatives.

Out of the 1 million of Web sites, 977,347 do not support HTTP/2. Table 4.6 and Figure 4.3 show for each of the possible errors the number of Web sites.

Reason	N	%
No HTTP/2 negotiated	604 857	61.89%
No HTTP/2 response	77 113	7.98%
No HTTPS (Refused)	129 549	13.26%
Timeout	136 510	13.97%
DNS	29 075	2.97%
Unknown	242	0.02%
Total	977 347	100.00%

Table 4.6: HTTP/2 over TLS failure reasons

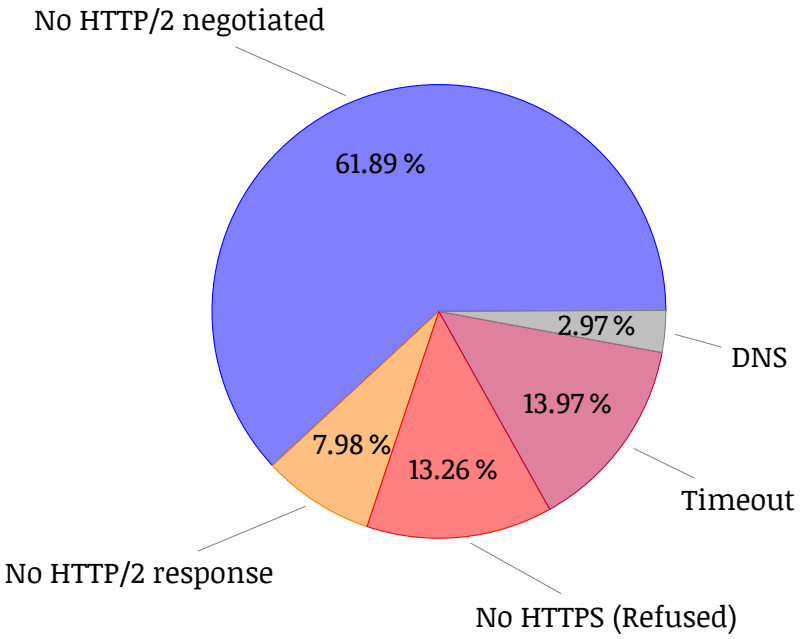

Figure 4.3: HTTP/2 over TLS failure reasons

The majority of the sites already indicate they do not support HTTP/2 during the TLS handshake. 77,113 servers do not indicate any protocol support during the TLS negotiation neither they respond to a HTTP/2 request.

Web sites do not necessarily need to implement a secure endpoint. 129,549 sites refuse connections to their port 443 and another 136.510 timed out in a manners time. It is interesting to note that the timeout is set to 3 seconds. This timeout is the main reason why it takes up to a week to collect the data.

This value is not expected before starting the crawling. 29,075 of the failed URLs, that represents the 2.97% of the failures, occur because the system is unable to resolve the domain.

A small list of Web sites failed due to unknown reasons (only 242 out of a 1 million). As it is a tiny percentage,it can be considered as a margin error.

4.2.2 HTTP Upgrade and Plain TCP

The URL's scheme does not change between the versions of the protocol. To discover if a server supports HTTP/2 over plain TCP, HTTP provides an upgrade mechanism to switch to a different protocol within the same connection.

In theory, HTTP clients should not make HTTP/2 requests directly to servers unless it is known from a previous connection it is supported. Some servers, like H2O, will respond to a direct HTTP/2 connection.

We check the support for the new protocol using the Upgrade method. And as a part of the tests, the script also makes a direct HTTP/2 request over plain TCP.

Table 4.7 shows the results.

	Upgrade			Plain TCP		
	Support	**Content**	**% Content**	**Support**	**Content**	**% Content**
100 000	0	0	-	0	0	-
200 000	1	0	0%	3	0	0%
350 000	2	1	50.00%	6	1	16.66%
600 000	5	3	60.00%	11	4	36.36%
1 000 000	16	10	62.50%	26	14	53.84%

Table 4.7: Implementation of HTTP/2 over plain TCP

The number of Web sites that support HTTP/2 using the Upgrade mechanism is very low.

In this cases, the reasons to do not server any content over plain TCP are the opposite to what happens over TLS: the response return a 3XX redirection to an https URL.

It is quite unexpected that the number of servers that support and also server content over plain TCP is bigger than using HTTP Upgrade mechanism, which is the way described by the RFC to start an HTTP/2 communication over non-secure channels (see Section 2.5).

All the 16 servers that support HTTP/2 via HTTP Upgrade also support HTTP/2 over TLS. In the case of plain TCP, 8 of 26 servers do not support HTTP/2 over TLS.

It is strange that comparatively many Web sites support HTTP/2 over TLS but not via HTTP Upgrade, neither over plain TCP. This mostly because their entry set up and/or load balancers redirects the traffic to different Web servers based on the entry port.

Table 4.8 shows the failure reasons and the number of Web sites considering both methods.

	Upgrade		Plain TCP	
Reason	**N**	**%**	**N**	**%**
No HTTP/2 response	940 171	94.02%	940 161	94.02%
No HTTP (Refused)	1 008	0.10%	1 008	0.10%
Timeout	34 969	3.50%	34 969	3.50%
DNS	23 620	2.36%	23 620	2.36%
Unknown	216	0.02%	216	0.02%
Total	999 984	100.00%	999 974	100.00%

Table 4.8: HTTP/2 over TCP failure reasons

We can conclude that as the major browsers, like Chrome and Firefox[11], only support HTTP/2 over TLS, it is affecting the way HTTP/2 is implemented across the Internet.

Chapter 5

HTTP/2 across the network

HTTP/2 will help the Web to deliver a faster and better user's experience with an improved usage of the network. Upgrading a protocol to a new version is not trivial, especially when there are billions of agents involved in the process. For a successful implementation, the majority of the users should benefit from it.

The Upgrade should only affect the endpoints, clients and servers, but that is not the case for HTTP. Today's Internet consists of a plethora of network entities, e.g., switches, routers, firewalls, NATs and proxies, that can alter HTTP requests and responses, especially headers, to provide their functionalities:

- Web provides to reduce the amount of data generated dynamically (e.g., traffic accelerators, cache).
- Traffic shaping (e.g., load load balancers).
- ISPs to provide a faster service and reduce their outbound traffic (e.g., traffic accelerators, cache, proxies).
- Optimizing the usage of IPv4 address space (e.g., NATs).
- Security (e.g., firewalls).

Those middleboxes can interfere the implementation of the HTTP/2. Protocols extensions are usually not designed considering that middleboxes could change new protocols design. Many new applications are deliberately designed to look like existent protocols or are actually tunneled over them to bypass middleboxes. This does not mean that it is impossible to deploy new protocols, but in order to ensure success it is imperative to first understand the interaction of the proposed solutions with the middleboxes along the path.

Recent studies on middleboxes behavior attempt to provide such information. However, the existing measurements use only a very small number of vantage points. Also, more

evidences of the nature of the problems and the portions of the network in which they manifest are needed.

In this chapter we quantify the impact of middleboxes with the deployment of the new protocol. We build a platform consisting of two clients and two servers, and we perform Internet measurements using a crowdsourcing platform. This approach allows us to recruit users to test the feasibility of the protocol considering its interaction with the elements of the path. In this work we use a low number of vantage points, but the methodology we implement has a great potential, since it is easily extendable.

5.1 Experimental Methodology and Setup overview

Some middleboxes process and modify the HTTP requests and responses to provide their functionality. How they will react to a new protocol with a completely different framing layer is unknown.

To answer the question, we use a crowdsourcing solution to collect information from many different paths that allows us study the potential interference of middleboxes. People from all over the world will help to test the feasibility and/or limitations of the implementation of HTTP/2.

For such task, we build a platform with one focus in mind: simplicity for the test users. The users do not necessarily have any technical background and, as external entities to the test, all the information should be produced and collected within the platform, with minimal requirements or input from them.

The platform is based on a client-server model, where both endpoints support HTTP/2 and they are able to establish a communication using the new protocol. Depending on the client network, this communication might not be possible and that circumstance must be recorded. We collect the data in the server side to check the number of successful connections, a simple subtraction indicates the number of failed client requests.

The platform consists of two clients applications (a Web application and an Android app) that connect to two Web servers, one encrypted and the other for plain communications. On the server, a PHP application provides the information to the clients and saves the data generated by them, storing it in a MySQL database for post-processing. Figure 5.1 shows a diagram of the platform.

The server provides a list of URLs to the clients with a combination of two domains (one secure and one non-secure) and 67 ports. The clients attempt to connect to all of

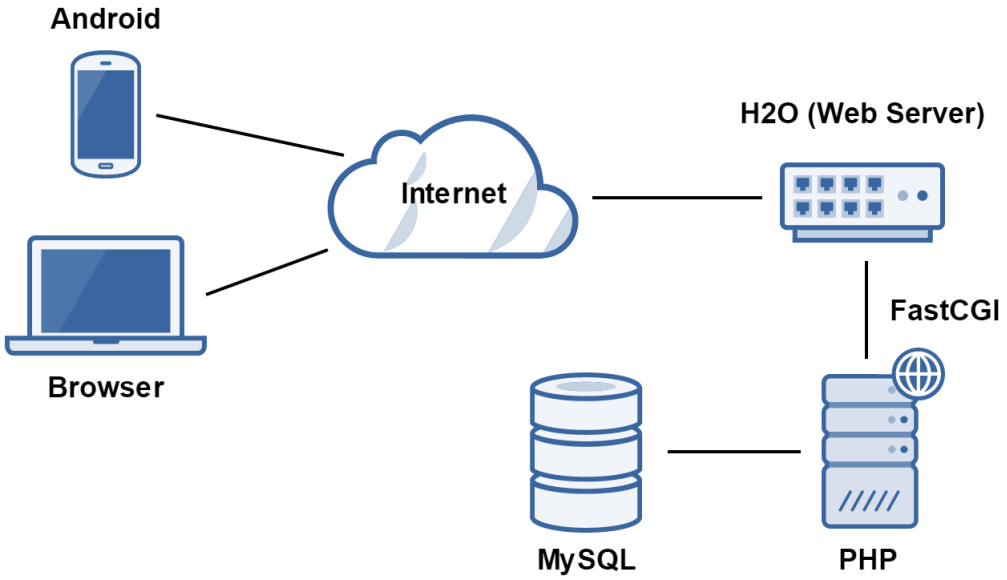

Figure 5.1: Platform setup

those tuples by first making an HTTP/1.1 request and then an HTTP/2 request. For the non-secure URLs, the Android client also attempts an HTTP/2 Upgrade.

The methodology used has a low number of vantage points, but this approach gives us the potential to easily extend it for different tests or changing on demand the servers and ports to test.

5.1.1 Crowdsourcing platform

Crowdsourcing platforms are an innovate online platforms that connect Employers and Workers from around the world. Employers create tasks to be completed by Workers, usually quick and simple, which are completed in a few minutes, thus they are called ”microjobs”.

Once the task has been defined by the Employer, a campaign is run in the platform with a specified number of times to be completed. Workers can apply to the campaign and complete the task. Each task contains a brief description of what needs to be done by the Workers and how the Employers will verify the completion of the task.

When the task has been completed as many of times the Employer required, the campaign is closed. Employers must verify that the task has been completed by the Workers and pay them accordingly. Crowdsourcing platforms are traditionally focused on the human element (i.e., to test psychological profile or to perform tests that machine

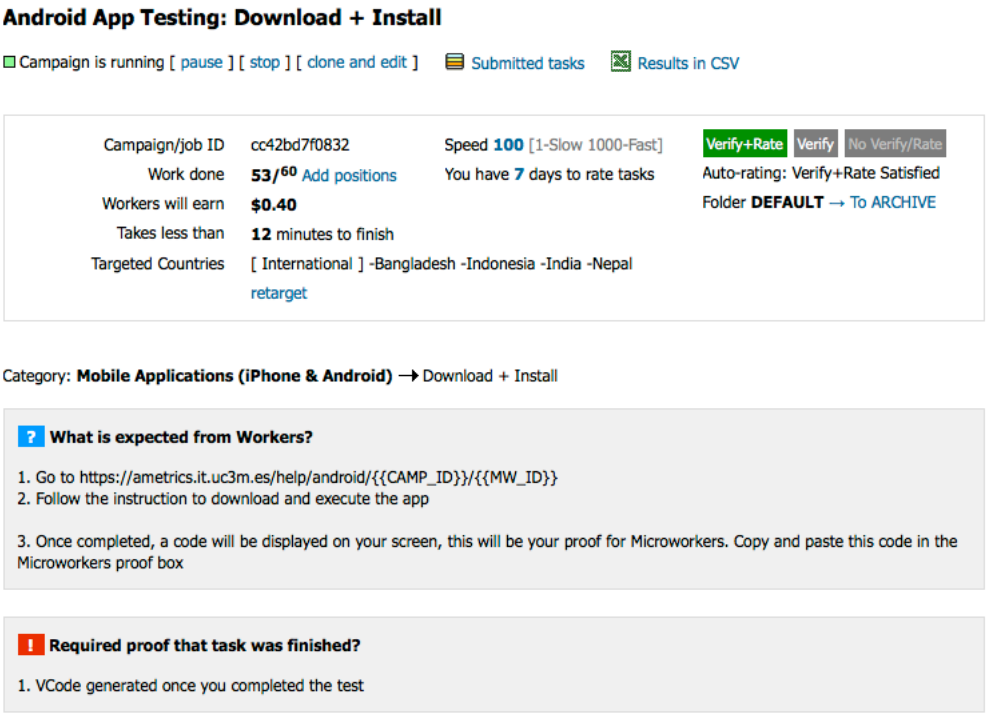

Figure 5.2: Microworkers campaign

could not do). We expand the usage of these platforms to run the Internet wide measurements.

We decide to use Microworkers as a crowdsourcing platform, due to different advantages in respect to other platforms (i.e. Amazon Mechanical Turks, MicroJob etc..):

- World-wide access to employers: Microworkers allows the workers to define own campaigns selecting the Countries in which the campaign will be run.
- Automatic payment method based on a unique verification code: to simplify this process Microworkers has a feature called VCODE, an automatic way to verify that the Worker has completed the task.
- The VCODE is a unique token generated by hashing the Campaign ID, the Worker ID and the Employer's secret key using SHA-256 algorithm.
- Possibility to select the MAs based on certain criteria, i.e. the type of Internet access (fixed or mobile) or even the type of measurement equipment used to perform the tasks.

Two campaigns are run in Microworkers, one for the browser tests and another for the Android application.

In the browser campaign, the Workers need to follow a link to the application. The Campaign ID and the Worker ID are auto-populated in the app, thanks to the available

substitutions in the job description. In the case of the Android application, the Workers follow a link to Google Play store to download the Android application. Then once opened, they introduce the Campaign ID and the Worker ID manually.

In both cases, the VCODE is generated by the server and returned once the task has been completed. The VCODE is displayed to the Workers, allowing them to copy it into the Microworkers and be paid once they have completed the task.

Figure 5.2 shows an example of the campaigns we create to recruit the users. Through Microworkers platform users are redirected to our web pages containing the instruction to perform the tasks in the case of Android (Figure 5.3). For the browser tests, they are redirected to the test page straight away.

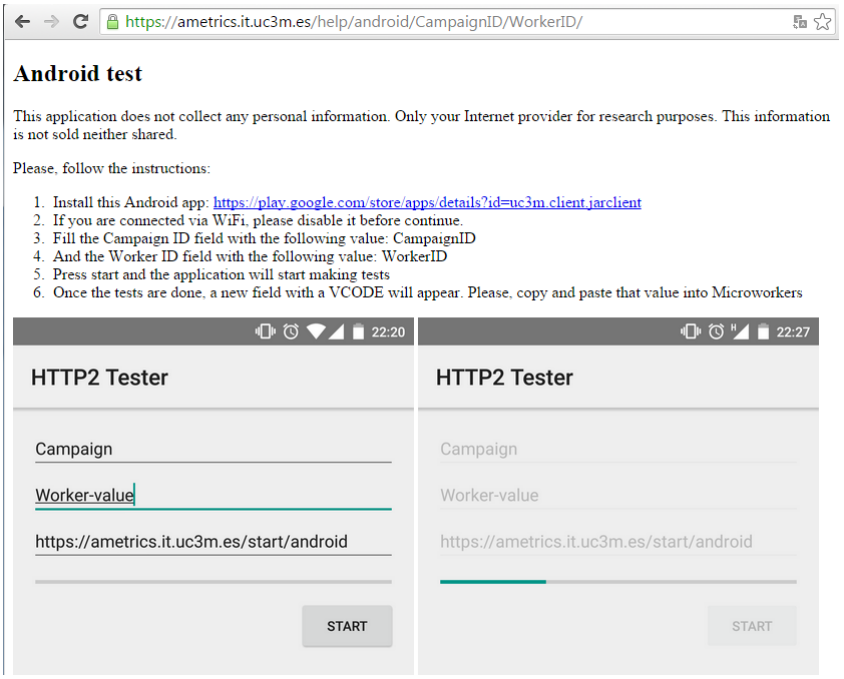

Figure 5.3: Instructions page for Android users

5.1.2 Measurement Server

As stated in Chapter 3, there are only two production ready servers: OpenLiteSpeed[20] and H2O[21]. Because of its simplicity and features, H2O is chosen for the platform. H2O is a general purpose Web server, it only provides an abstracting layer between HTTP and the back-end. It supports HTTP/1.1, HTTP/2 and also HTTP/2 Upgrade over plain TCP.

Two servers are installed with an identical configuration. The only difference between both installations is the usage of encrypted connections, with OpenSSL used to provide TLS support. Attached to them through FastCGI, a PHP application provides the client

applications all the information needed for the tests and also collects all the information produced by them. This information is stored into a MySQL database for the post-processing phase.

The application consists on 3 endpoints in a REST like service: start, test and finish, which respond with JSON content. Figure 5.4 shows the interaction between the client and the server.

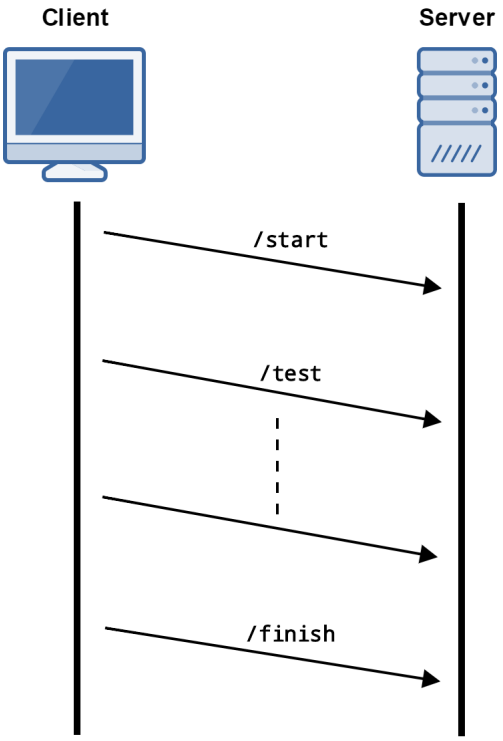

Figure 5.4: Server endpoints diagram

Clients make a request to the start endpoint to obtain a list of URLs to perform the test on. These URLs point to the test endpoint with different combination of hosts and ports. Once the test is completed, the application confirms it by making a finally request to finish endpoint.

The endpoints provide the following functionality:

- `/start/{client}/({campaign}/{worker}/)?`
  This endpoint expects a client string in the URL and an optionally a JSON in the payload with extra information. It returns a JSON with an unique token and a list of URLs to perform requests on. This token is used to distinguish between different requests produced by each full test. It accepts two optional parameters in the URL, the Campaign ID and the Worker ID as an integration part for Microworkers.
- `/test/{token}/(upgrade/)?`
  The tests are performed against this endpoint, it receives the token as part of the

URL and returns a JSON. If the request is correct, the JSON confirms it and also returns the protocol used. For the HTTP/2 Upgrade requests, the URL endpoint contains the Upgrade string in it.

This endpoint saves all the information about the request into the database: remote IP address, remote port, HTTP method, protocol (HTTP/1.1 or HTTP/2), scheme (http or https), hostname, port, if the request is an Upgrade, HTTP headers, etc.

- `/finish/{token}/`
  The last endpoint only receives the token as part of the URL and it marks the test as completed for that token. New requests against the server using this token are discarded. If the start request contains a Campaign ID and a Worker ID, this endpoint returns a JSON with the VCODE for the verification of the task.

The reason to have a different URL for the HTTP Upgrade test endpoint is because the Upgrade is transparent to the PHP application, everything is handled by H2O. The server passes `"HTTP/1.1"` or `"HTTP/2"` strings as server protocol variable to PHP. This only way to distinguish if the test is a direct request or an attempt of an HTTP/2 Upgrade.

Start and finish endpoints are requested using the default ports for HTTP communications. For a better support, the server of preference for the clients to request those endpoints is the server using encrypted HTTP.

5.1.3 Browser Client

The browser client is a simple Web application built using PHP and JavaScript hosted in H2O along with the back-end application. The application makes requests only to secure URLs using AJAX, as majors browser have only support for H2. Figure 5.5 shows a screenshot of the initial state.

The Webapp accepts two extra parameters as a part of the URL path: the Campaign ID and the Worker ID, they are auto-populated in the start endpoint URL. Users only need to click on *Start* button to begin the test.

After clicking on ”start”, the application makes a request to the start endpoint. The server checks in the start point if the protocol used is HTTP/2 to make sure that the browser has support for it. If there is no support for it, an alert box is shown indicating such thing.

The application iterates over each URL making an AJAX requests to the server. A progress bar reports to the user about the status of the test. As AJAX requests are sent asynchronously, to avoid overwhelming the server there is a 400 ms interval in between the requests.

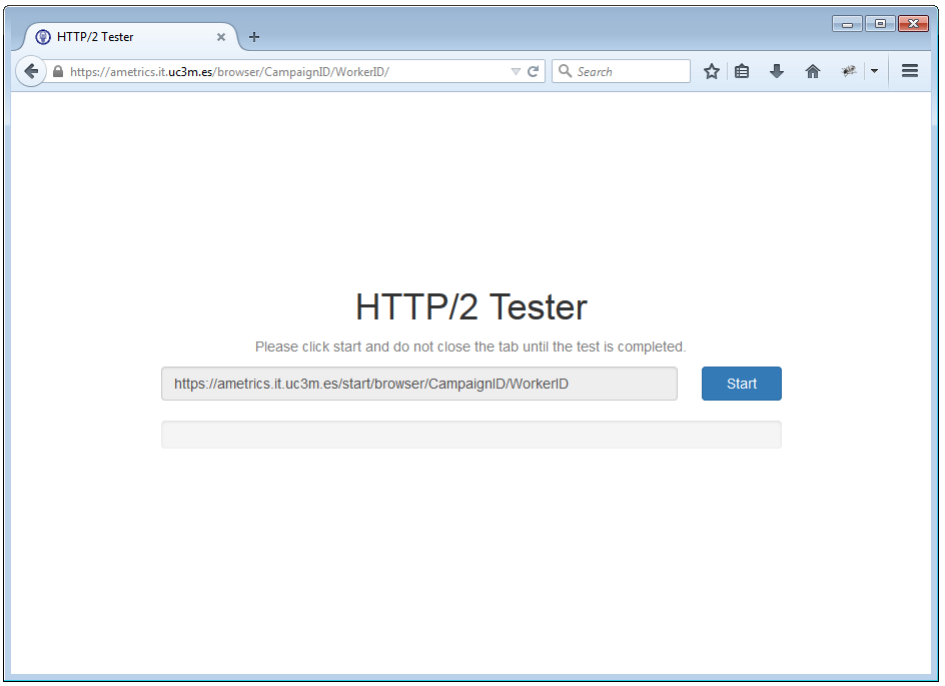

Figure 5.5: Browser application

The server application returns the protocol used by H2O as part of the JSON response. If the protocol matches with HTTP/2, an internal counter is increased.

Once the tests are completed, the application shows an alert box with the number of test performed and the number of successful tests. It also displays the VCODE allowing Workers to paste it into Microworkers, Figure 5.6.

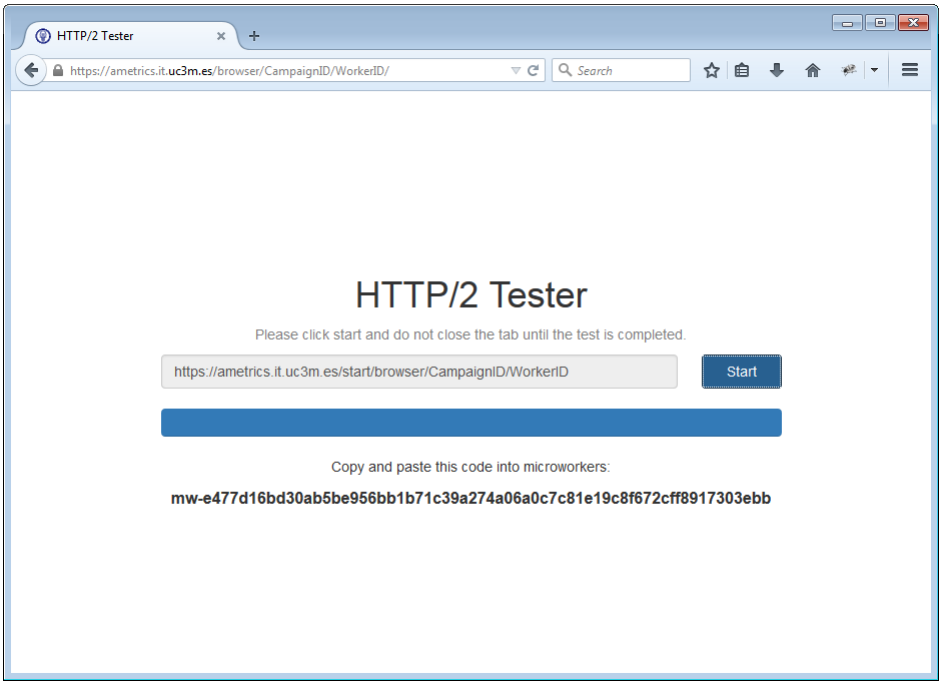

Figure 5.6: Browser application on completed

5.1.4 Android Client

Mobile networks usually have more middleboxes in their paths. To test these networks, a mobile application is developed for Android devices using the language Java. In this case, we do not have the limitations of the browsers. The application performs an HTTP/1.1, an HTTP/2 and an HTTP/2 Upgrade -only over plain TCP- request to each URL.

To avoid the low level details of the protocol, we use the library OkHttp[2]. OkHttp is an HTTP client with support for HTTP/1.1 and HTTP/2. Because of a potential bug in the ALPN extension in Android, the library OkHttp is modified to make a direct HTTP/2 requests over TLS without protocol negotiation (see Section 5.1.5).

OkHttp does not have support for HTTP/2 Upgrade. To perform those tests, we use a TCP socket and implement a custom HTTP client. The application sends the plain HTTP Upgrade using HTTP/1.1 and reads the response. If the response has a 200 status code, that means no Upgrade was performed. If the response has a 101 status code, the server is accepting the HTTP/2 Upgrade. Before the server sends the HTTP/2 response, the client sends the preface as confirmation, an HTTP/1.1 like message with the following content:

```
PRI * HTTP/2.0\r\n\r\n
SM\r\n\r\n
```

After that, the client sends two hard-code binary messages to the server with the client settings and an acknowledgement to the `SETTINGS` frame sent by the server.

The implementation of HTTP/2 in any language is outside of the scope of this project. The previous HTTP/2 communication is implemented into the application to allow the server to send the response. Without the acknowledge to the `SETTINGS` frame, the server waits for 3 seconds until the response is sent and the connection closed because of a protocol violation. This partial implementation works correctly with H2O server, it does not necessarily work with other HTTP/2 servers.

The interface of the application is very similar to the browser implementation. Figure 5.7 shows the initial view of the application. In this case, two text input fields are added to allow the user to introduce the details related to the implementation for Microworkers: Campaign ID and Worker ID. Once those input boxes are filled, the users press the button *Start*.

The application performs two tests, one mandatory test over mobile networks and, if available, another test using a WiFi connection. The logic described in the next paragraphs is the same in both cases.

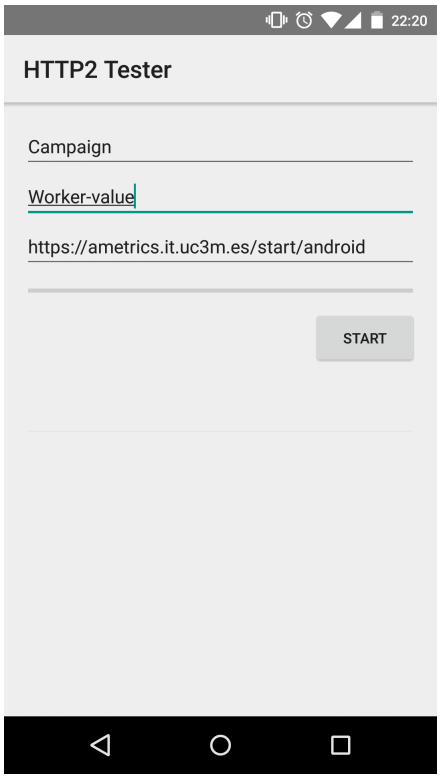

Figure 5.7: Initial page of the application

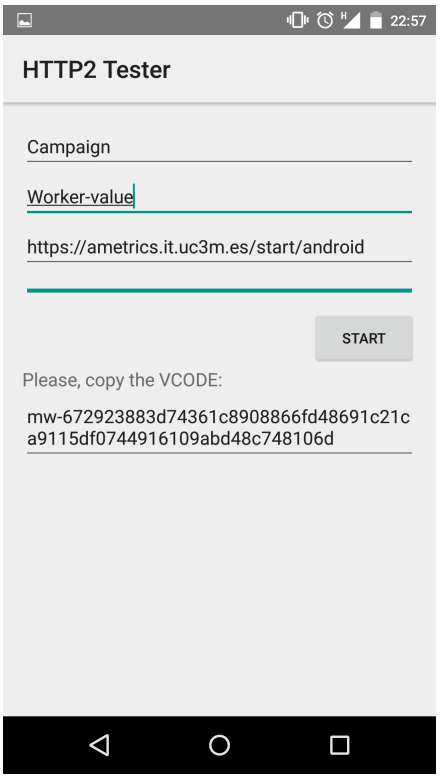

Figure 5.8: After complete the tests

After clicking on ”start”, the application makes a request over HTTPS to the start endpoint to get the list of URLs. If the test is over mobile networks, the application sends extra information about the connection: local IP, network type and subtype, mobile country code, mobile network code and cell ID.

For each URL, the application makes an HTTP/1.1, an HTTP/2 Upgrade -plain TCP only- and an HTTP/2 requests. A progress bar shows the status of the test to the users.

If the request is responded by the server with a 200 status code, the application increases a counter to show the results at the end. In the case of HTTP/2 Upgrade, the application also reads the JSON response looking for the protocol used which returned by the server to confirm that HTTP/2 is being used.

Once the tests are completed for mobile connections, the application check for WiFi connection. If available, the client repeats the process for this new connection.

On completion of all test, the application shows an alert box with the results of the tests and displays the VCODE allowing Workers to paste it into Microworkers, Figure 5.8.

5.1.5 Limitations

There are a number of limitations that needs to be considered during the development of the platform.

For the browser client, Chrome, Firefox and Opera[11] only support HTTP/2 over TLS. This client only performs tests to secure URLs, it does not make any request using HTTP Upgrade mechanism, neither over plain TCP. Also for security reasons, browser blocks connection to ports 20 and 25 unless it is explicitly set by the users. Requests on those ports for browser tests are not included in the results.

In TLS, the negotiation is done during the handshake using the extension ALPN. This extension has limited support in Android, it is only supported in Android 5 or above. There is support for it in Android 4.4, but it does not work correctly because of a bug[43]. This is one of the reasons to choose H2O as it supports HTTP/2 communication without a previous negotiation for both types of communication, secure and non-secure.

The market share of Android 5.0+ is just 18%[44]. In order to not be restricted with the supported devices, OkHttp library is modified to remove the ALPN negotiation. With this change, the Android application works in devices with the versions 4.4 and above, which represents about 58% of the market.

5.2 Data sets

For the measurements, 658 workers are involved: 336 workers participate to the browser based campaign, while 322 workers participate to the application-based one. Out of these 322 workers, only 49 of them havve both mobile and WiFi connectivity available, and thus generate fixed line results (over WiFi).

In the browser-based campaign, each worker attempts two connections (HTTP/1 and H2) per port against our servers. In the Android application based campaign, each worker attempts four connections for each one of the 67 ports to test: HTTP/1, H2 without ALPN, H2C and H2C without Upgrade.

Figure 5.9 shows how mobile users are partitioned based on network type. Figure 5.10 shows a more detailed view of the different subtypes of 3G network.

Overall, workers are distributed among 38 countries as shown in Figure 5.11.

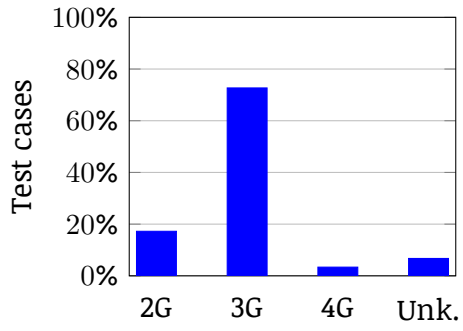

Figure 5.9: Tests by mobile network type

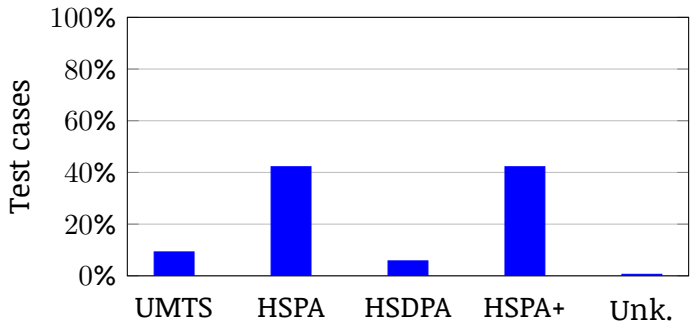

Figure 5.10: Tests by 3G network subtype

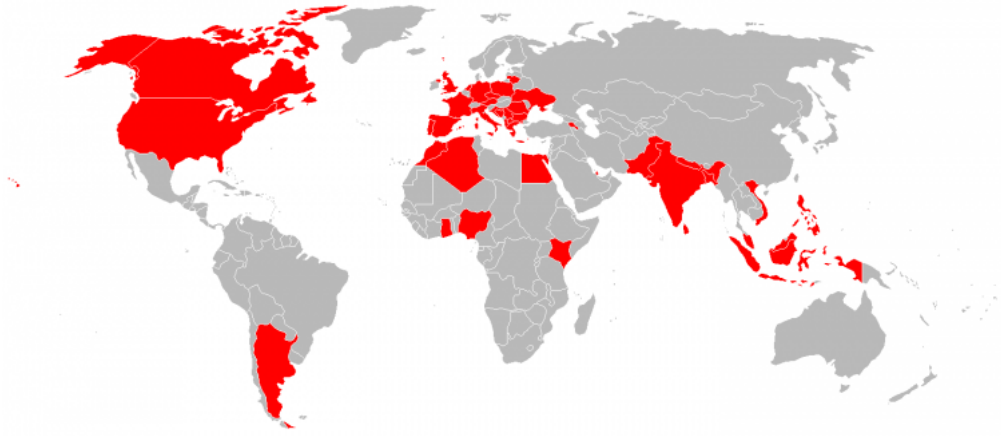

Figure 5.11: Distributed vantage points map

5.3 Results

To obtain the results, we calculate the number of request completed for HTTP/1.1 for each port and subtract the number of completed requests for HTTP/2. HTTP/2 requests are only evaluated if the HTTP/1.1 requests are successful for the same type of connection: TLS or plain TCP. That gives us the number of errors for HTTP/2 compared to HTTP/1.1. In the case of HTTP/2 Upgrade, not all the requests upgrade to HTTP/2. Requests that do not upgrade to the new protocol are considered as errors.

5.3.1 Fixed line

In this subsection, we show the results for fixed line. First, we present the percentage of workers that are not able to perform an H2 connection using their own browser; next we show the percentage of errors when workers use the Android application over WiFi connection.

Figure 5.12 shows the error rate as a function of the port number; these results were collected via regular browser using fixed access. On average, the error rate is 2%, an expected number as only encrypted communications were used.

We measure a high error rates for the following ports: 80 (4.90%), 593 (6.90%) and 5554 (5.55%). There are well-known ports used for Web, remote procedure call over HTTP and SGI ESP HTTP, respectively. We can deduce from those numbers that, likely, middleboxes are used to monitor or prevent traffic on these given ports. On the default port for encrypted HTTP (443), the error rate is 0%.

Figure 5.13 shows the error rates for H2 without ALPN, H2C and H2C without Upgrad. These results were produced via our Android application using WiFi connectivity.

Overall, we do not detect many errors. Over port 80 we measure an error rate of 2.04% for both types of unencrypted connections: Upgrade and without Upgrade. There is a 100% success rate for encrypted communications on the default port 443.

5.3.2 Mobile networks

Figure 5.14 shows the error rate as a function of the port number. In this case we differentiate three type of tests: H2 without ALPN, H2C and direct. These results are generated via our Android application using mobiles connectivity.

Overall, the error rate is quite low on average: 0.58%. When compared with the results for connection over WiFi, the error on port 80 raises up to 7% for HTTP/2 Upgrade. These results are remarkable, since this is the most common usage of unencrypted HTTP/2. The error rate reduces to 4.4% when attempting a direct unencrypted connection.

It seems that the adoption of HTTP/2 Upgrade will face additional challenges when compared with the encrypted connection, where the error rate is just 0.33%.

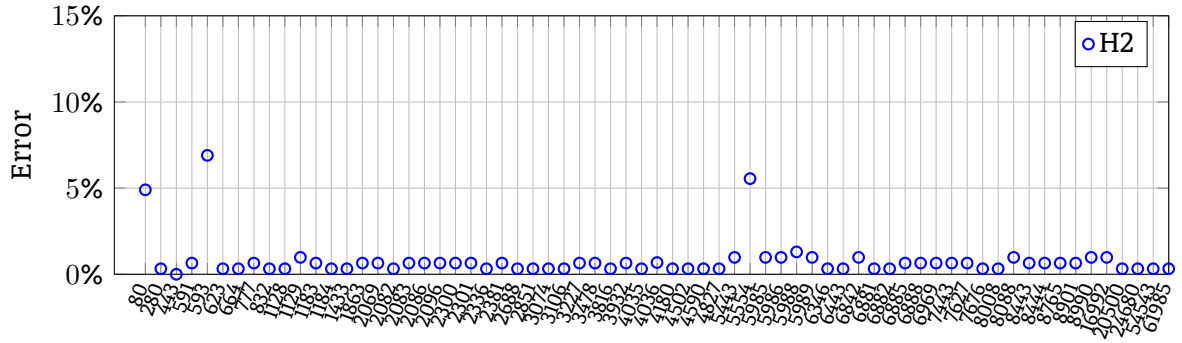

Figure 5.12: Error rate vs. port, browsers (fixed line)

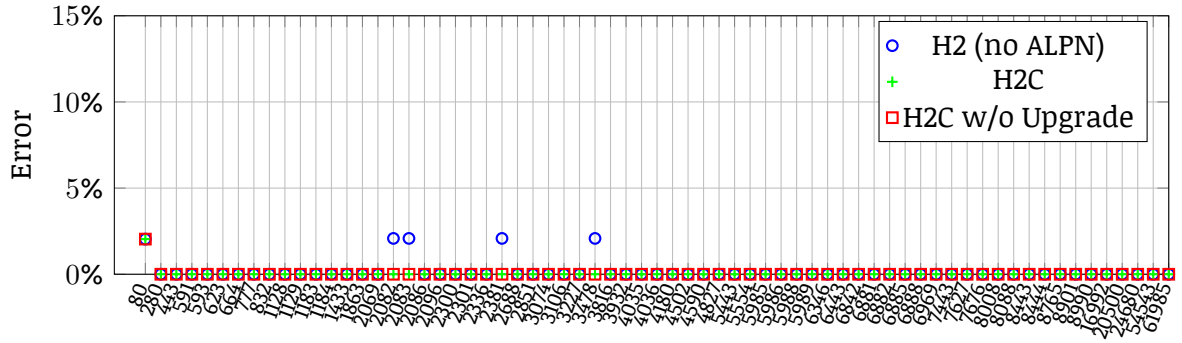

Figure 5.13: Error rate vs. port, Android WiFi (fixed line)

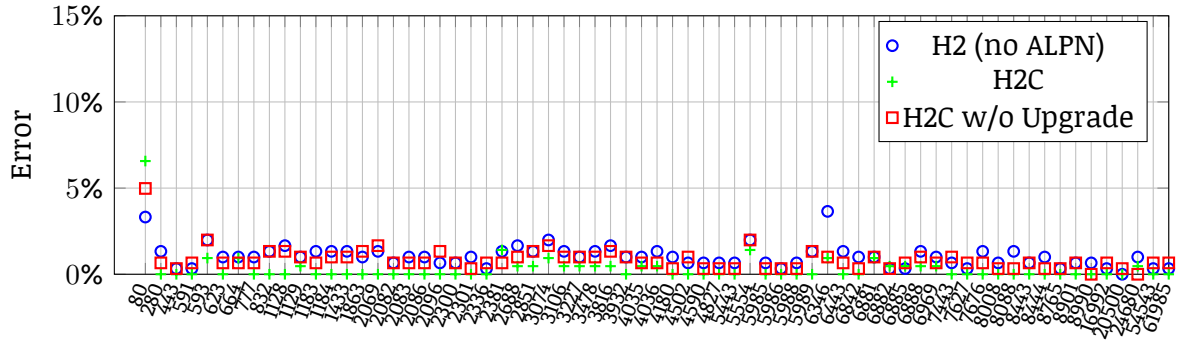

Figure 5.14: Error rate vs. port, mobile networks

5.3.3 Proxies

In order to better understand the results in mobile environment, we extend our study by analyzing proxy behaviour over port 80. We consider the HTTP headers of the workers that produced errors for H2 without ALPN, H2C and direct and we try to identify the use of proxies along the path.

Proxies can be divided into two groups: anonymous and non-anonymous. Anonymous proxies are not detectable by servers, non-anonymous proxies leave traces of their presence in the HTTP headers.

To discover which workers are behind of proxies, we check for the existence of the following headers in the HTTP requests:

- `CLIENT-IP`
- `FORWARDED`
- `FORWARDED-FOR`
- `FORWARDED-FOR-IP`
- `PROXY-CONNECTION`
- `VIA`
- `X-FORWARDED`
- `X-FORWARDED-FOR`
- `X-GATEWAY`
- `X-NETWORK-INFO`

Table 5.1 shows the percentage of workers that produce an error and are behind a proxy, for H2 without ALPN, H2C and H2C without Upgrade in mobile network case studies. Considering the case of H2C, results show that 24% of errors are due to proxies. A 40% of these requers do not arrive to our sever and the remaining 60% do not Upgrade to HTTP/2. When the workers send the H2C request without the Upgrade, the percentage of errors due to a proxy is 12.5%.

Based on these results, a direct H2C request without Upgrade is more successful than a H2C connection. In this case, proxies do not recognize the Upgrade and change the HTTP/1.1 101 Switching Protocols header to a normal HTTP/1.1 request.

Protocol	Number of errors	Number of proxies
H2 w/o ALPN	3.32%	18%
H2C	6.6%	24%
H2C w/o Upgrade	5%	12.5%

Table 5.1: Android-based mobile campaign proxy errors

5.3.4 Carrier-grade NAT

Carrier-grade NAT (CGN), also known as large-scale NAT (LSN), is an approach to IPv4 network design in which end users are configured with private network addresses that are translated to public IPv4 addresses by middlebox network address translator devices

embedded in the network operator's network, permitting the sharing of small pools of public addresses among many end sites.

To detect which test cases are behind carrier-grade NAT, we count the number of different IP addresses that are used for each unique token. 29 test cases are found, but we exclude 1 worker because it is already behind a proxy. For these cases, 75 different IPv4 addresses are used.

The results are shown in Figure 5.15. For H2 without ALPN connections, there are no errors in most of the ports. Only in port 80 we detect a significant error rate, 3 workers are not able to perform an H2 connection.

In the case of H2C, the Upgrade is always successful except for one worker, which it is not able to perform a H2C connection. In this case, not only port 80 is affected.

When Upgrade is not used, we detect the same error rate as in the case of H2C. The same worker in this case is not able to perform an H2C connection directly to the server. One more error has been detected on port 80.

Overall, results demonstrate that NATs does not influence the deployment of H2 on the Internet.

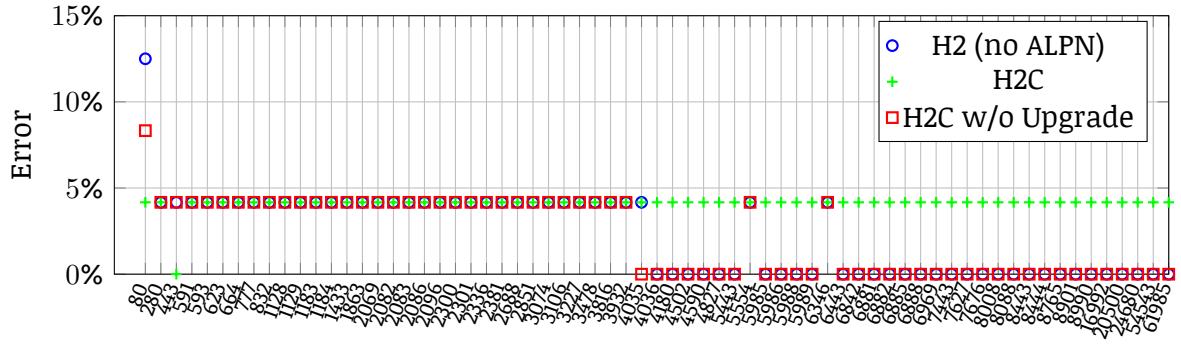

Figure 5.15: Error rate vs. port, NATs connections

Chapter 6

Related Work

We identified two main research areas related to our work: HTTP/2 studies and measurements of adoption of new protocols.

HTTP/2 studies focus on understanding its adoption and performance [45], [46]. Saxce et al.[45] take on the challenge of understanding whether HTTP/2 provide performance improvements compared to HTTP/1. Their main outcome is that, within a lab environment and using synthetic content, HTTP/2 provides only minor performance improvements compared to HTTP/1, and that it can largely suffer in lossy environments.

Differently from [45], Varvello et. al [46] study both HTTP/2 adoption and performance in the wild. Apart from showing how large corporations adopt HTTP/2, [46] shows that popular Web design techniques, like domain sharding and inlining, mitigate HTTP/2 limitations within lossy environments. Our work differs from both previous works[45], [46] since we focus on HTTP/2 interaction with the current Internet ecosystem rather than measuring its performance or adoption.

A broader research corpus is available when studying the deployment of a new protocol on the Internet. Here we select a subset of works that share some similarities with our work [47], [48], [49]. Honda et al. [47] shed some light on the feasibility of extending TCP due to its (bad) interaction with middleboxes. Authors demonstrate that middleboxes are almost omnipresent and that they can arbitrarily change packets header (e.g., dropping both known and unknown TCP options) or payloads, particularly over port 80. However, the authors only have access to a low number of vantage points (49 residential, 34 Hotspot, 20 cellular, 17 University and 17 Enterprice networks). To overcome the latter issue, Hirth et al. [49] demonstrate that crowdsourcing platforms can become a powerful tool to achieve a realistic view of the network from an end-user perspective. In other work [48], they also show the benefits of using a crowdsourcing platforms

to gather realistic vantage point at Internet scale. Specifically, they leverage such technique to study TLS interaction with deployed middleboxes. In this work, we extend such methodology to test HTTP/2.

Chapter 7

Conclusion

This work presents the details of the implementation of HTTP/2, software support, deployment across the Internet and the interaction between HTTP/2 and the Internet ecosystem.

The software support is on track. Major browsers already support HTTP/2 which helps to the deployment of the protocol.

Our crawler tests the HTTP/2 support using the Alexa top 1 million websites as sample. The results show that, as of October 2015, already 22,653 websites announce support for HTTP/2 over TLS, but only 10,162 of them -45%- show content to the users over HTTP/2. The support for HTTP/2 over clear TCP (Upgrade) is minimal, with just 16 servers with support for it, with only 10 websites providing content over it.

In this work, we also present the results about HTTP/2 interaction with the network through a crowdsourcing campaign. To study the interaction between HTTP/2 and the Internet ecosystem, we build a measurement platform to test whether a given network location can use all HTTP/2 communication methods as specified in its RFC: encrypted HTTP/2 and unencrypted HTTP/2 using the Upgrade mechanism. We also check for direct unencrypted HTTP/2 connection, without using the upgrade.

Overall, we tested 4 variations accessing 67 different ports from 38 countries across the world. Our results suggest that the presence of middleboxes on the Internet can affect H2C deployment, especially if the server is listening on port 80 and the client uses a mobile network, with a 7% of failure. In the case of HTTP/2 over TLS connection, there is a 100% success rate on the default port 443.

Chapter 8

Quote

This project details the HTTP/2 protocol, the software support, the implementation across the Internet and the behavior of the network with the new protocol.

The realization of this project is divided in the following phases and sub-phases:

- Explanation of the HTTP/2 protocol.
- Software support research.
- Implementation across the Internet.
  - Development of the platform.
  - Analysis of the data.
- Network behavior.
  - Development of the server.
  - Development of the browser client.
  - Development of the Android client.
  - Analysis of the data.

The realization of the project took 8 weeks, 5 weeks of research, analysis and writing and 3 weeks of writing.

As external equipment required, two servers for a month were needed.

The following table describes the cost of the project:

Description	**Duration**	**Cost**	**Total**
Engineer	8 weeks	500€/week	4,000€
VPS	2x4 weeks	10€/week	80€
		Subtotal	4,080€
		IVA (20%)	816€
		Total	**4,896€**

Table 8.1: Quote of the project

The total cost of the project is four thousand eight hundred ninety-six euro (4,896€).

Leganés, 5 of October of 2015.

Engineer's signature:

Printed name: José Fernando Calcerrada Cano

Abbreviations

AJAX	**A**synchronous **J**avaScript **a**nd **X**ML
ALPN	**A**pplication-**L**ayer **P**rotocol **N**egotiation
CGI	**C**ommon **G**ateway **I**nterface
CSV	**C**omma-**S**eparated **V**alues
DNS	**D**omain **N**ame **S**ystem
HTTP	**H**yper**t**ext **T**ransfer **P**rotocl
IESG	**I**nternet **E**ngineering **S**teering **G**roup
IETF	**I**nternet **E**ngineering **T**ask **F**orce
IP	**I**nternet **P**rotocol
ISP	**I**nternet **S**ervice **P**rovider
JSON	**J**ava**S**cript **O**bject **N**otation
NAT	**N**etwork **A**ddress **T**ranslation
NPN	**N**ext **P**rotocol **N**egotiation
RFC	**R**equest **f**or **C**omments
SHA	**S**ecure **H**ash **A**lgorithm
TCP	**T**ransmission **C**ontrol **P**rotocol
TLS	**T**ransport **L**ayer **S**ecurity
URL	**U**niform **R**esource **L**ocator
VPS	**V**irtual **P**private **S**erver
WWW	**W**orld **W**ide **W**eb

Bibliography

[1] M. Belshe, R. Peon, and M. Thomson. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540, May 2015. URL http://www.ietf.org/rfc/rfc7540.txt.

[2] OkHttp - An HTTP+SPDY client for Android and Java applications, . URL https://github.com/square/okhttp. Accessed: 2015-07-12.

[3] M Hirth, T Hoßfelda, M Melliac, C Schwartza, and F Lehriedera. Crowdsourced network measurements: Benefits and best practices, 2015.

[4] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, June 1999. URL http://www.ietf.org/rfc/rfc2616.txt.

[5] B. Kahle. HTTP Archive - Trends, June 2015. URL http://httparchive.org/trends.php. Accessed: 2015-06-30.

[6] StatCounter. Comparison from jan 2009 to aug 2015 | statcounter global stats, June 2015. URL http://gs.statcounter.com/#comparison-ww-monthly-200901-201507. Accessed: 2015-07-02.

[7] I. Grigorik. High-Performance Browser Networking, 2013. URL http://chimera.labs.oreilly.com/books/1230000000545/ch11.html#HTTP11_MULTIPLE_CONNECTIONS. Accessed: 2015-06-20.

[8] M. Belshe. Research Blog: A 2x Faster Web, November 2009. URL http://googleresearch.blogspot.co.uk/2009/11/2x-faster-web.html. Accessed: 2015-07-02.

[9] C. Bentzel. Chromium Blog: Hello HTTP/2, Goodbye SPDY, February 2015. URL http://blog.chromium.org/2015/02/hello-http2-goodbye-spdy-http-is_9.html. Accessed: 2015-07-07.

[10] E. Protalinski. Mozilla outlines Firefox roadmap for HTTP/2, February 2015. URL http://venturebeat.com/2015/02/18/mozilla-outlines-firefox-roadmap-for-http2/. Accessed: 2015-07-07.

[11] D. Stenberg. TLS in HTTP/2, March 2015. URL http://daniel.haxx.se/blog/2015/03/06/tls-in-http2/. Accessed: 2015-07-07.

[12] Networking/http2 - MozillaWiki, . URL https://wiki.mozilla.org/Networking/http2. Accessed: 2015-07-07.

[13] R. Trace and D. Walp. HTTP/2: The Long-Awaited Sequel - IE Blogs, October 2014. URL http://blogs.msdn.com/b/ie/archive/2014/10/08/http-2-the-long-awaited-sequel.aspx. Accessed: 2015-07-07.

[14] F. Olivier. Building a more interoperable Web with Microsoft Edge, June 2015. URL http://blogs.windows.com/msedgedev/2015/06/17/building-a-more-interoperable-web-with-microsoft-edge/. Accessed: 2015-07-07.

[15] C. Benfield. HTTP/2 Picks Up Steam: iOS 9, June 2015. URL https://lukasa.co.uk/2015/06/HTTP2_Picks_Up_Steam_iOS9/. Accessed: 2015-08-30.

[16] Usage Statistics and Market Share of Web Servers for Websites, July 2015, July 2015. URL http://w3techs.com/technologies/overview/web_server/all. Accessed: 2015-07-05.

[17] S. Eissing. mod_h2 - HTTP/2 for Apache httpd. URL https://github.com/icing/mod_h2. Accessed: 2015-07-06.

[18] O. Garrett. How NGINX Plans to Support HTTP/2, February 2015. URL https://www.nginx.com/blog/how-nginx-plans-to-support-http2/. Accessed: 2015-07-06.

[19] N. Lala. HTTP/2 for IIS in Windows 10 Technical Preview, October 2014. URL http://blogs.iis.net/nazim/http-2-for-iis-in-windows-10-technical-preview. Accessed: 2015-07-06.

[20] S. Espitia. LiteSpeed Web Server: The World's First Web Server to Offer HTTP/2 Support, May 2015. URL http://blog.litespeedtech.com/2015/05/20/litespeed-web-server-the-worlds-first-web-server-to-offer-http2-support/. Accessed: 2015-07-06.

[21] O. Kazuho. H2O HTTP/2 server version 1.3.0 released, June 2015. URL http://blog.kazuhooku.com/2015/06/h2o-http2-server-version-130-released.html. Accessed: 2015-07-06.

[22] G. Wilkins. Introduction to HTTP2 in Jetty, May 2015. URL https://webtide.com/introduction-to-http2-in-jetty/. Accessed: 2015-07-09.

[23] Understanding the ALPN API, June 2014. URL http://www.eclipse.org/jetty/documentation/current/alpn-chapter.html. Accessed: 2015-07-09.

[24] D. Stenberg. The State and Rate of HTTP/2 Adoption, May 2015. URL http://daniel.haxx.se/blog/2015/03/31/the-state-and-rate-of-http2-adoption/. Accessed: 2015-07-01.

[25] HAProxy - News, May 2015. URL http://www.haproxy.org/news.html. Accessed: 2015-07-10.

[26] RoadMap - Squid Web Proxy Wiki, May 2015. URL http://wiki.squid-cache.org/RoadMap. Accessed: 2015-07-10.

[27] R. Okubo. HTTP/2 Support - Apache Traffic Server, February 2015. URL https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=51810161. Accessed: 2015-07-10.

[28] hitch TLS proxy, . URL https://github.com/varnish/hitch. Accessed: 2015-07-20.

[29] T. Tatsuhiro. nghttpx, . URL https://nghttp2.org/documentation/nghttpx.1.html. Accessed: 2015-06-20.

[30] S. Ludin. With HTTP/2, Akamai Introduces Next Gen Web, February 2015. URL https://blogs.akamai.com/2015/02/with-http2-akamai-introduces-next-gen-web.html. Accessed: 2015-07-10.

[31] HTTP/2 Wireshark, . URL https://wiki.wireshark.org/HTTP2. Accessed: 2015-07-11.

[32] Network monitor - firefox developer tools | mdn. URL https://developer.mozilla.org/en-US/docs/Tools/Network_Monitor. Accessed: 2015-09-24.

[33] Evaluating network performance - google chrome. URL https://developer.chrome.com/devtools/docs/network. Accessed: 2015-09-24.

[34] HttpWatch, . URL http://www.httpwatch.com. Accessed: 2015-07-11.

[35] B. Fitzpatrick. h2i interactive HTTP/2 console debugger, . URL https://github.com/bradfitz/http2/tree/master/h2i. Accessed: 2015-07-11.

[36] D. Stenberg. cURL Changelog - Fixed in 7.38.0, September 2014. URL http://curl.haxx.se/changes.html#7_38_0. Accessed: 2015-07-12.

[37] T. Tatsuhiro. nghttp, . URL https://nghttp2.org/documentation/nghttp.1.html. Accessed: 2015-07-12.

[38] G. Molnár and N. Hurley. node-http2 - An HTTP/2 client and server implementation for node.js. URL https://github.com/molnarg/node-http2. Accessed: 2015-07-12.

[39] S. Ludin. http2-perl - Perl implementation of the HTTP/2.0 protocol. URL https://github.com/sludin/http2-perl. Accessed: 2015-07-12.

[40] B. Fitzpatrick. http2 - HTTP/2 support for Go, . URL https://github.com/bradfitz/http2. Accessed: 2015-07-12.

[41] C. Benfield. hyper - HTTP/2 for Python. URL https://github.com/lukasa/hyper. Accessed: 2015-07-12.

[42] I. Grigorik. http-2 - Pure Ruby implementation of HTTP/2 protocol. URL https://github.com/igrigorik/http-2. Accessed: 2015-07-12.

[43] E. Anderson. Don't use broken ALPN on Android 4.4 | github. URL https://github.com/square/okhttp/issues/1305,note=.

[44] Lollipop now running on 18.1% of Android devices according to official numbers for August 2015. URL http://phandroid.com/2015/08/03/android-platform-versions-august-2015/. Accessed: 2015-08-10.

[45] H. de Saxce, I. Oprescu, , and Yiping Chen. Is http/2 really faster than http/1.1? Computer Communications Workshops (INFOCOM WKSHPS), 2015 IEEE Conference on, May 2015.

[46] M. Varvello, K. Schomp, D. Naylor, J. Blackburn, A. Finamore, and K. Papagiannaki. To http/2, or not to http/2, that is the question, 2016.

[47] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend tcp? pages 181–194, 2011.

[48] A. M. Mandalari, M. Bagnulo, and A. Lutu. Informing protocol design through crowdsourcing: the case of pervasive encryption. August 2015.

[49] M. Hirth, T. Hoßfeld, M. Mellia, C. Schwartz, and F. Lehrieder. Crowdsourced network measurements: Benefits and best practices. 90.